Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Angewandte Informatik

Bachelor-Arbeit
zur Erlangung des akademischen Grades
Bachelor of Science - B.Sc.

Implementierung künstlicher neuronaler Netze auf einem Xilinx FPGA der Zynq-Serie

vorgelegt von Harald Heckmann

am 30. September 2016

Referent: Prof. Dr. Steffen Reith
Korreferent: Prof. Dr. Adrian Ulges

Erklärung gem. ABPO, Ziff. 6.4.3

Ich versichere, dass ich die Bachelor-Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 30.09.2016 Harald Heckmann

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Arbeit:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger	√	
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	√	
Veröffentlichung des Titels der Arbeit im Internet	√	
Veröffentlichung der Arbeit im Internet	√	

Wiesbaden, 30.09.2016 Harald Heckmann

Für Nina Pizewitsch

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
APU	Application Processing Unit
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Network
DSP	Digital Signal Processor
EMIO	Extended Multiple Use Input Output
FIFO	First in First Out
FLOPS	Floating-Point Operations
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
II	Initiation Intervall
IP-Core	Intellectual Property Core
KNN	Künstliches Neuronales Netz
LUT	Lookup Table
MACC	Multiply-Accumulate
MSB	Most Significant Bit
MUX	Multiplexer
OCM	On-Chip Memory
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
ROM	Read-Only Memory
SCU	Snoop Control Unit
SDSoC	Software Defined System on a Chip
SOC	System on a Chip
XSDK	Xilinx Software Development Kit

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Quellcodes

Zusammenfassung

Diese Arbeit evaluiert die Möglichkeiten, Effizienz und Effektivität der Implementierung eines „Künstlichen Neuronalen Netzes" (KNN) auf einem „Field Programmable Gate Array" (FPGA). Die Besonderheit der Implementierung zeigt sich darin, dass ein komplettes Software-/Hardware Co-Design, welches auf einem einzigen FPGA betrieben wird, in einer semantisch stark abgewandelten Version der Programmiersprache C entwickelt wird.

Kapitel 1

Einleitung

In dieser Arbeit wird ein „Künstlichen Neuronalen Netz" (KNN) auf einem „Field Programmable Gate Array" (FPGA) implementiert. Die Besonderheit hierbei liegt darin, dass die diese in einer semantisch stark abgeänderten Version der Programmiersprache C erstellt wird.

1.1 Stand der Forschung

Zunächst wurden KNN auf Computern in sequentiellen Prozessoren ausgeführt. Schon gegen Ende der 80er Jahre wurden KNN auf speziell dafür angefertigter Hardware realisiert [Rü96]. Seitdem wurden sie nicht nur auf sequentiellen Prozessoren ausgeführt, sondern auch als speziell dafür angefertigte Hardware in Form analoger und digitaler „Application-Specific Integrated Circuits" (ASICs) implementiert. Seit der Jahrtausendwende wurden neben sequentiellen Prozessoren und ASICs auch Graphics Processing Unit (GPU) und FPGA als grundlegende Hardware für Implementierungen von KNN verwendet. Die Entwicklung dieser auf GPU wurde durch die general-purpose Programmierschnittstelle „Compute Unified Device Architecture" (CUDA) ermöglicht, welche im Jahre 2007 von Nvidia veröffentlicht wurde. Ein Beispiel einer solchen Implementierung, die kurz nach der Veröffentlichung von CUDA publiziert wurde, ist unter [JPJ08] zu finden.
Es existieren mehrere Konzepte zur Umsetzung von KNN auf Hardware, im folgenden werden zwei vorgestellt:

1. Klassisch mit Knoten und Kanten als tatsächliche Hardwareeinheiten

2. Reihe von Matrixoperationen

Im Jahre 2006 wurde eine Methode präsentiert, die schildert wie KNN auf FPGA in der klassischen Vorstellung realisiert werden können. Die Besonderheit der Methode liegt darin, dass eine Reduktion der Kanten (und somit der Leitungen) durch die Virtualisierung dieser Kanten erreicht wurde [OR06]. Ebenso wurden die Ergebnisse bereits als Reihe von Matrixoperationen auf FPGA berechnet. Ein Hardwaredesign, welches die Ergebnisse als Reihe von Matrixoperationen berechnet, wurde im Jahr 2015 vorgestellt und zeichnet sich durch einen großen Datendurchsatz aus [GAGN15]. Diese Implementierungen wurden in einer Hardwarebeschreibungssprache erstellt. Bei der Entwicklung in einer Hardwarebeschreibungssprache gilt es viele Einzelheiten zu beachten, was lange Entwicklungszeiten verursachen kann. Dementsprechend kann die Änderung bestehender Designs viel Zeit beanspruchen.

1.2 Motivation

Die Implementierung des KNN in einer Hardwarebeschreibungssprache, das Dokumentieren des Vorgehens und die Evaluierung der Schwierigkeiten standen zu Beginn dieser Arbeit im Mittelpunkt. Ein Vergleich mit bestehenden Implementierungen sollte die Effizienz der entstandenen Implementierung hervorheben. Aus im Kapitel 4 genannten Gründen wird die Entwicklung jedoch nicht in einer Hardwarebeschreibungssprache, sondern in C durchgeführt.
Seit etwa 1998 ist es möglich, Hardwareschaltkreise in einer höheren Programmiersprache, beispielsweise C, oder einem Subset dieser zu entwickeln [GS98]. Eine besondere Form dieser Übersetzungswerkzeuge erstellt für Hybridsysteme, bestehend aus einem oder mehreren Prozessoren und einer programmierbaren Logik, ein Hardware/Software Co-Design. Diese Übersetzungswerkzeuge erstellen die erforderlichen Dateien für den Prozessor oder die Prozessoren und die programmierbare Logik und generieren Code, der nach den Angaben des Entwicklers den Datenfluss zwischen diesen Komponenten (siehe SDSoC [Xil16a]) steuert. Sie werden kontinuierlich verbessert und von mehreren Hochschulen und Unternehmen parallel entwickelt. Seit einiger Zeit hat die Effektivität und Effizienz dieser Übersetzungswerkzeuge einen Grad erreicht, der es ermöglicht, diese erfolgreich kommerziell einzusetzen. Eine aktuelle Studie (2016) listet die bekannten Übersetzungswerkzeuge auf und vergleicht diese [NSP+16]
Die Entwicklung in einer höheren Programmiersprache gewährt zusätzliche Flexibilität

und verkürzt die Entwicklungszeiten maßgeblich. Die Motivation hat sich insofern geändert, dass die Implementierung in einer höheren Programmiersprache (C) durchgeführt wird. Dabei wird evaluiert, wie effizient und effektiv die Art und Weise der Implementierung als auch das fertige Produkt sind.

1.3 Ziel

Das Ziel ist es, ein schnelles KNN in einer höheren Programmiersprache, in diesem Fall C, zu entwickeln und einem FPGA auszuführen. Das KNN soll Eingabewerte in Ausgabewerte umwandeln. Ein Training des KNN ist in der Implementierung nicht vorgesehen. Die Implementierung wird so erstellt, dass das KNN bedingt konfigurierbar und somit nicht komplett statisch ist. Die Implementierung soll sich schnell ändern lassen, sodass beispielsweise eine andere Topologie für das neuronale Netz relativ schnell implementiert werden kann. Dabei soll evaluiert werden, ob die Entwicklungsumgebung „Software Defined System on a Chip" (SDSoC) von Xilinx für solche Implementierungen geeignet ist und welche Vor- und Nachteile sich bei der Verwendung von SDSoC ergeben. Idealerweise wird die Implementierung auf dem FPGA annähernd so schnell laufen wie Implementierungen von vergleichbaren KNN auf einer im Jahr 2016 aktuellen GPU, jedoch wesentlich weniger Strom verbrauchen, was eines der Hauptargumente für die Verwendung von FPGAs für eine solche Anwendung darstellen könnte.

Kapitel 2

Grundlagen I: Field Programmable Gate Arrays

„Field Programmable Gate Arrays" sind Chips, die zum Aufbau digitaler logischer Schaltungen dienen. Die Besonderheit liegt hierbei in der Möglichkeit, diese digitalen logischen Schaltungen auf dem Chip zu programmieren, den FPGA somit zu rekonfigurieren. FPGA sind programmierbare Hardware. Auf den Aufbau der FPGA, Hybrid-FPGA mit Prozessoreinheiten und die Verwendung von FPGA wird in diesem Kapitel eingegangen.

2.1 Aufbau

FPGA sind nicht die einzigen Chips, die zum Aufbau digitaler logischer Schaltungen dienen und konfigurierbar sind. Es gibt viele unterschiedliche Ansätze zum Aufbau solcher Chips [Wan98], hier wird jedoch nur der Aufbau von FPGA beschrieben.
Der rekonfigurierbare Teil des FPGA heißt „Programmable Logic" (PL). Dieser Teil besteht hauptsächlich aus vielen „Configurable Logic Block"s (CLB) und einer geschickten Verdrahtung. Durch die Verdrahtung können mehrere Logikblöcke zusammengefasst werden, was die Realisierung komplexer Funktionen ermöglicht.
Neben weiteren notwendigen Komponenten, wie beispielsweise Taktpuffern, welche Taktflanken mit hoher Leistung im Schaltkreis verteilen, gibt es noch optionale Komponenten. Dies sind zum Beispiel „Block Random Access Memory" (BRAM) Blöcke, welche zur Erhöhung des verfügbaren Speichers mit geringeren Zugriffszeiten dienen, und „Digital Signal Processor" (DSP) Blöcke, die zur schnellen Ausführung bestimmter arithmetischer Operationen zur Verfügung stehen. Auf den grundlegenden Aufbau von

FPGA und diesen drei Komponenten, CLB, BRAM und DSP wird in diesem Kapitel eingegangen.

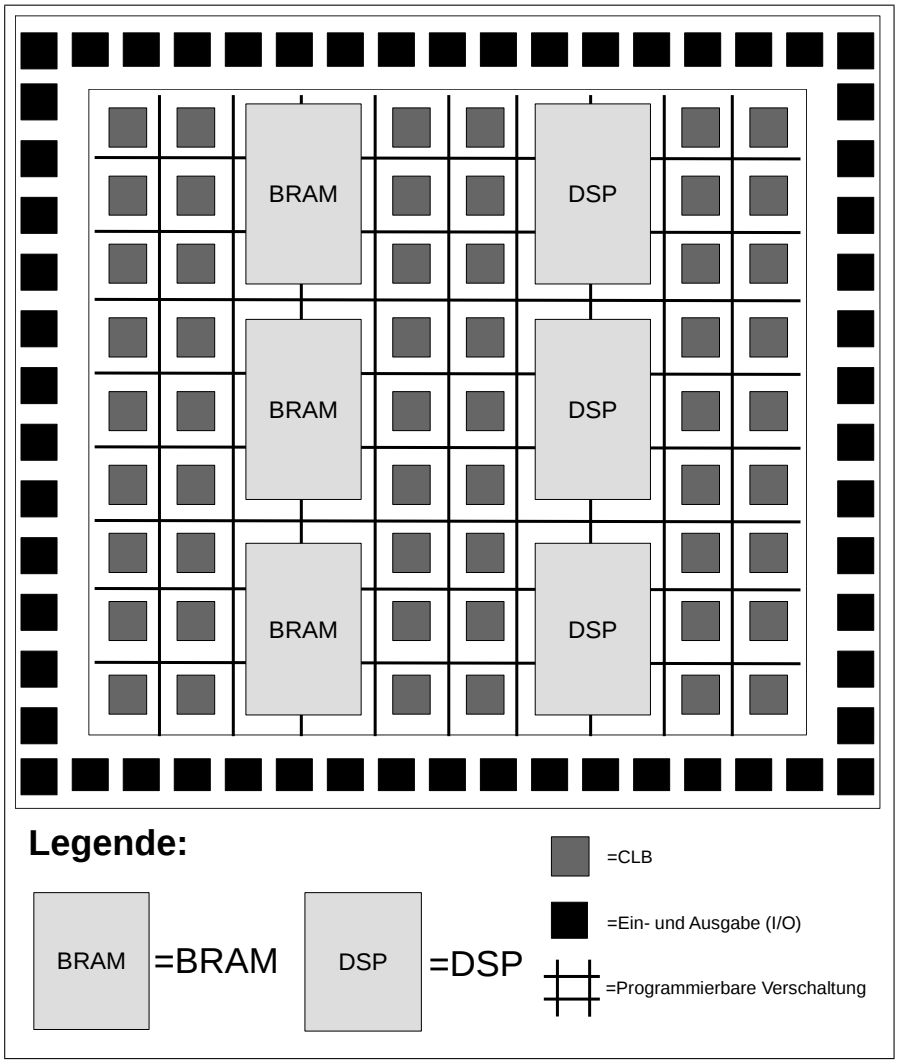

Abbildung 2.1: Stark vereinfachte Darstellung der programmierbaren Logik eines FPGA (Vergleich: [Wan98, Sau10])

Die stark vereinfachte Darstellung der PL aus Abbildung 2.1 enthält, bis auf die Ein- und Ausgabeblöcke, die für diese Arbeit relevanten Komponenten. Die CLB, BRAM Blöcke und DSP Blöcke sind für das Projekt dieser Arbeit von zentraler Bedeutung und werden im Folgenden erklärt. Die Leitungen im FPGA und die Verschaltung der einzelnen Komponenten sind Bestandteile von FPGA, die in dieser Arbeit nicht weiter erklärt werden.

2.1.1 Configurable Logic Block (CLB)

Der CLB ist ein grundlegender Logikbaustein in FPGA, der es ermöglicht das logische Verhalten der digitalen Schaltung zu bestimmen. Auf FPGA von Xilinx werden CLB in jeweils zwei „Slices" unterteilt. Diese Logikblöcke bestehen aus den folgenden Bausteinen [Sau10, Xil14b]:

- „Lookup Table" (LUT), eine Wahrheitswertetabelle
- Konfigurierbare 1-Bit Speicher, die D-FlipFlops oder Latches sein können
- „Multiplexer" (MUX) zur Auswahl des Ausgabesignals

Die Anzahl der genannten Bausteine kann in FPGA von unterschiedlichen Herstellern variieren. Die LUT ist eine Wahrheitswertetabelle und bildet boolesche (logische) Funktionen ab. Üblicherweise wird eine Zahl zusätzlich zur LUT genannt, welche die Anzahl der Adressleitungen (Eingangsleitungen) angibt. So ist der Bezeichnung $LUT_n; n \in \mathbb{N}^+$ zu entnehmen, dass es sich um eine LUT mit n Adressleitungen (Eingangsleitungen) und einer einzigen Ausgabeleitung handelt, womit 2^n Eingabewerte jeweils auf einen Ausgabewert abgebildet werden. Dies ist eine n Bit auf 1 Bit (wahr/falsch) Abbildung. Es ist auch möglich die Anzahl der Ausgabewerte zu variieren. Eine $LUT_{n\_m}; n, m \in \mathbb{N}^+; n \geq m$ bildet n Bit auf m Bit ab, wird jedoch in FPGA nicht so häufig wie eine LUT verwendet die n Bit auf 1 Bit abbildet.
Das D-FlipFlop dient zur Speicherung des letzten Ausgabewertes der LUT. Soll explizit keine Ausgabe der LUT verlangt werden, so kann der gespeicherte Wert der letzten Ausgabe wiederverwendet werden. Die Selektion zwischen dem gespeicherten Wert und dem Ausgabewert der LUT wird mit einem Multiplexer erreicht.

2.1.2 Speicher

Die D-FlipFlops in den CLB können als, zur Taktflanke synchrone, 1-Bit Speicher verwendet werden, die durch Zusammenschluss zu Registern größere Datenmengen Speichern können. Da nicht viele Flip-Flops vorhanden sind, eignen sie sich nicht für die Speicherung größerer Datenmengen. Üblich sind Kapazitäten bis zu 128 Bit. Je kürzer der Signalweg ist, umso kürzer sind die Signallaufzeiten. Register befinden sich unmittelbar an den Stellen, an denen Daten benötigt werden. Aus diesem Grund sind Signallaufzeiten gering. Durch den schnellen Zugriff auf Daten in den Registern, der jedoch

geringen Kapazität über den gesamten FPGA, werden diese primär für flüchtige Daten verwendet. D-FlipFlops werden auf Grund der genannten Eigenschaften hauptsächlich als Speicher in Pipelines und für Zwischenergebnisse, wie beispielsweise das Ergebnis einer Berechnung oder Speicheranfrage, verwendet. Die LUT Bausteine in CLB können als Speicher verwendet werden. Xilinx nennt diese Form von Speicher distributed RAM (verteilter RAM). Werden nicht nur synchrone Schreibzugriffe, sonder auch synchrone Lesezugriffe benötigt, muss zusätzlich zu den LUT ein Flip-Flop aus den CLB verwendet werden. Distributed RAM ist schneller Speicher. Dieser kann in unmittelbarer Nähe des Funktionsblocks instantiiert werden, in dem er verwendet wird, da die CLB über den gesamten FPGA verteilt sind. Auf Grund der Nähe von distributed RAM zu den Funktionsblöcken, in denen dieser benötigt wird, sind die Signallaufzeiten sehr gering. Dieser Speicher ist nur in geringer Menge vorhanden und eignet sich nicht, um größere Datenmengen zu speichern. Für kleine, persistente Datenmengen eignet sich der Speicher gut. Einige Verwendungsszenarien sind in [Xil14b] dargestellt. Der Begriff des „Block Random Access Memory" (BRAM) beschreibt einen zusammenhängenden Speicher. BRAM Blöcke in Xilinx FPGA sind folgendermaßen aufgebaut [Xil14c]: Sie haben eine Größe von 36 Kilobit (RAMB36E1) und können jeweils auf zwei mal 18 Kilobit (RAMB18E1) große BRAM Blöcke aufgeteilt werden. Sie können sowohl als FIFOs, als Fehlerkorrektur „Random Access Memory" (RAM) oder als Allzweck RAM bzw. Read-Only Memory (ROM) verwendet werden. Diese BRAM sind „true dual port RAM", was bedeutet, dass Lese- und Schreiboperationen auf jeweils zwei Ports mit zwei (optional phasenverschobenen) Takten möglich sind.

2.1.3 Digital Signal Processor (DSP) Block

Ein DSP Block ist ein Baustein, welcher in digitalen Schaltungen verwendet wird, um Signale geordnet umzuwandeln. Auf einem FPGA dienen die DSP Blöcke der schnellen Berechnung arithmetischer Ausdrücke. Die DSP Blöcke auf Xilinx FPGA können die folgenden Operationen schnell durchführen [Xil16b]: **Multiplikation**, Addition, Subtraktion, **Akkumulierung**, Shiften und Mustererkennung. Die DSP Blöcke werden bei neueren FPGA von Xilinx DSP48E1 genannt und können 25 mal 18 Bit multiplizieren und 48 Bit akkumulieren [Xil14a]. Interessant für die Implementierung des KNN ist hierbei, dass diese DSP „Multiply-Accumulate" (MACC) Operationen in einem Taktzyklus durchführen können.

2.2 Xilinx Zynq

Die Zynq-Serie der Xilinx FPGA hat die Besonderheit, dass sich neben der PL das sogenannte „Processing System" (PS) befindet. Das PS enthält mindestens zwei ARM Cortex9a Prozessoren. Der gesamte FPGA bildet somit ein „System on a Chip" (SoC) [LHCS14, Xil15].
Folgende Abbildung stellt einen FPGA mit einer PS und einer PL, sowie deren Kommunikationskanäle dar.

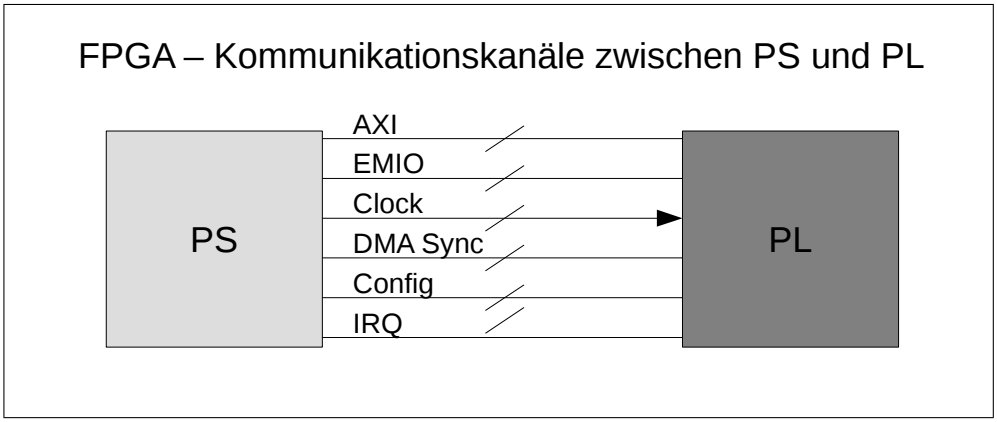

Abbildung 2.2: Kommunikationskanäle zwischen PS und PL (Vergleich: [Xil15])

Im weiteren Verlauf dieses Kapitels werden die für diese Arbeit relevanten Inhalte des PS dargestellt und erklärt, wie über feste Kommunikationskanäle mit einem „Advanced eXtensible Interface" (AXI) Bus ein effizienter Datenaustausch zwischen PS und PL stattfindet.

2.2.1 Processing System (PS)

Das PS befindet sich unmittelbar neben der PL auf dem selben Chip. Üblicherweise wird das PS als Kontrolleinheit verwendet, welche die PL zur Beschleunigung rechenintensiver Operationen verwendet. Der wichtigste Teil des PS ist die „Application Processing Unit" (APU), welche unter Anderem aus mindestens 2 ARM Cortex9a Prozessoren und einer „Snoop Control Unit" (SCU) besteht. Diese kontrolliert die Kohärenz der Cachespeicher zwischen den ARM Prozessoren und dem L2 Cache. Die SCU ermöglicht außerdem, dass die PL Zugriff auf die Cachespeicher hat und den Inhalt dieser mit den Prozessoren teilt. Der Rest des PS besteht aus verschiedenen Controllern, beispielsweise aus einem

DDR2/DDR3/LPDDR2 Speichercontroller, Ein- und Ausgabeeinheiten, Verschaltungseinheiten und Schnittstellen, wie z.B. Clocks, Interrupts oder AXI Bus Schnittstellen, auf die im Kapitel „Kommunikation über das AXI Bussystem" genauer eingegangen wird [LHCS14]. Es folgt eine vereinfachte Darstellung des PS mit den für die Kommunikation relevanten Bausteinen. Eine genauere Darstellung ist in [Xil15] zu finden:

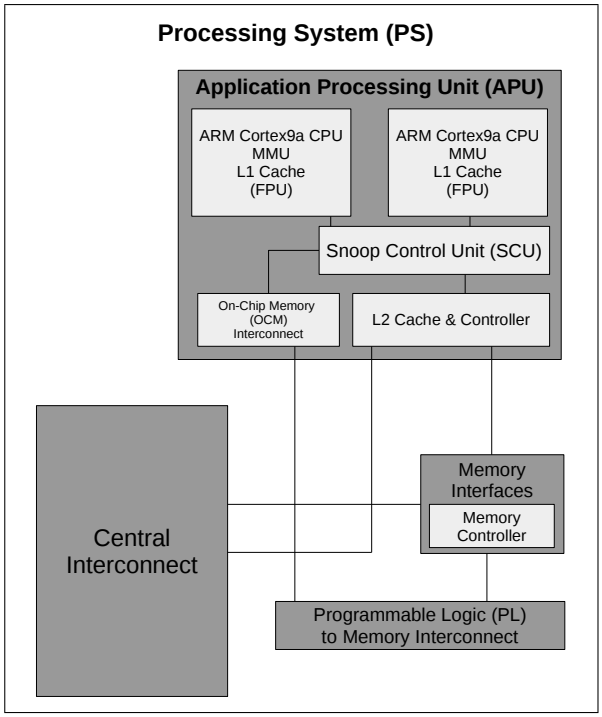

Abbildung 2.3: Vereinfachte Darstellung einiger Komponenten des PS (Vergleich: [Xil15])

2.2.2 Programmierbare Logik (PL)

Die PL wird bei hybriden FPGA, wie den FPGA der Xilinx Zynq-Serie, normalerweise als Beschleuniger verwendet. Dafür werden, je nach Anwendungsfall, unterschiedliche AXI Komponenten auf der PL benötigt. Diese werden benötigt, um die Kommunikation mit dem PS zu steuern und AXI Signale korrekt auf der PL zu verwalten. AXI Komponenten auf der PL können beispielsweise „First In First Out" (FIFO) Puffer, Verschaltungseinheiten (Interconnect) zur Selektion der aktiven Teilnehmer des AXI Busses oder Systeme zum Zurücksetzen der AXI Komponenten in einen definierten Zustand (Reset System) sein. Werden AXI Komponenten benötigt, können diese, falls noch nicht vorhanden, in einer Hardwarebeschreibungssprache beschrieben werden. Des Weiteren werden die AXI

Komponenten nur instantiiert, wenn diese benötigt werden. Das heißt, die Verwendung des AXI Bussystems ist auf der PL sehr flexibel und zugleich ressourcensparsam, da eine Vielfalt an Implementierungen existiert, jedoch nur die notwendigen Implementierungen instantiiert werden und Ressourcen verbrauchen.

2.2.3 Kommunikation über das AXI Bussystem

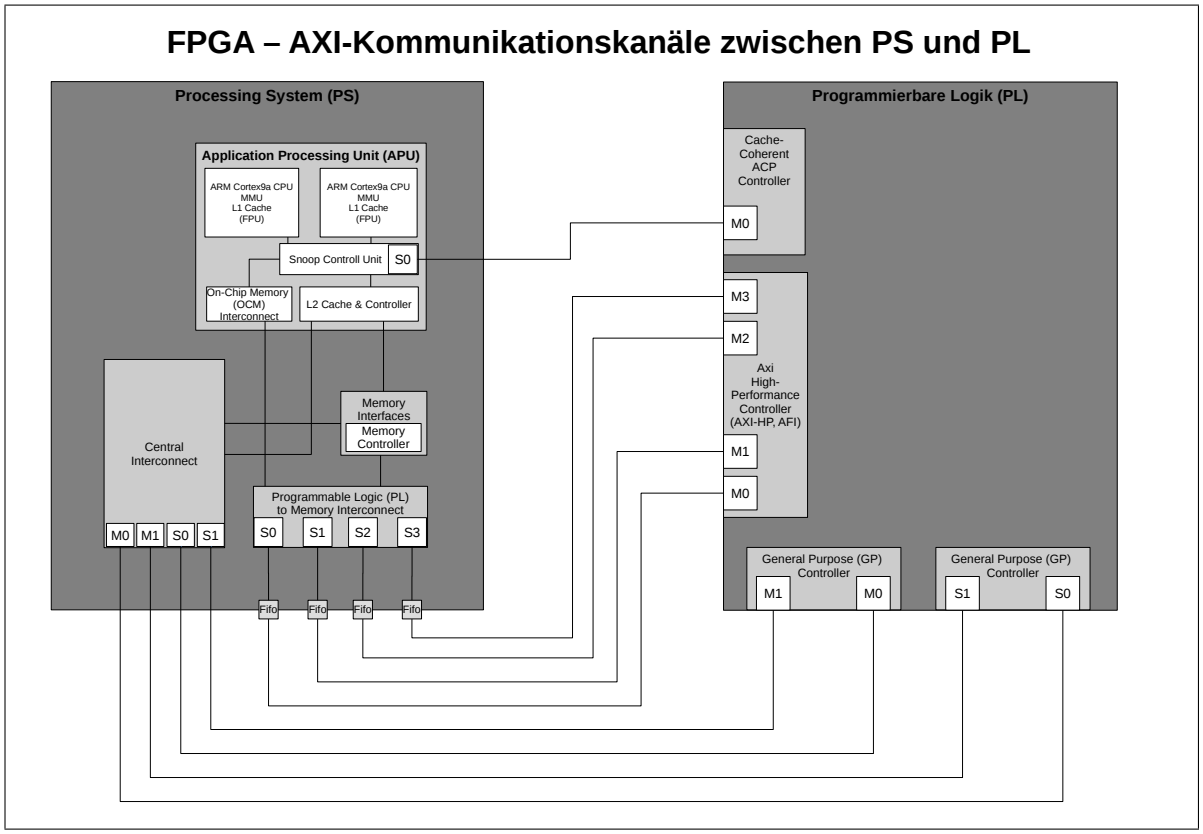

Abbildung 2.4: Vereinfachte Darstellung der AXI-Kommunikationskanäle zwischen PS und PL (Vergleich: [Xil15, LHCS14])

Die universale Kommunikationsschnittstelle zwischen PL und PS ist ein AXI Bussystem, bestehend aus verschiedenen zueinander asynchronen Kommunikationskanälen [Xil15, Arm11]. Für unterschiedliche Verwendungszwecke gibt es unterschiedliche AXI Schnittstellen:

Die cache-kohärente AXI ACP Schnittstelle hat die Besonderheit, dass die PL Daten über die SCU vom 512 Kilobyte großen, assoziativen Acht-Wege-L2 Cache und über den

Kanal	Kanalanzahl	Master	Slave	Verwendung
AXI General Purpose	Zwei	PL	PS	Kleine Datenmengen, Skalare
AXI General Purpose	Zwei	PS	PL	Kleine Datenmengen, Skalare
AXI Cache-Coherent ACP	Einer	PL	PS	Mittelgroße Datenmengen
AXI-HP / AFI	Vier	PL	PS	ab mittelgroßen Datenmengen

Tabelle 2.1: AXI-Kommunikationskanäle in FPGA der Zynq-Serie

„On-Chip Memory Controller" (OCM) auch vom 256 Kilobyte großen „Static Random-Access Memory" (SRAM) abrufen und verändern kann. Dabei erhält die SCU die Cache-Koherenz mit den beiden L1-Caches der beiden ARM Cortex9a Prozessoren aufrecht. Das bedeutet, dass die Daten des L2-Cache und des SRAM mit den Daten des L1-Cache synchronisiert werden. Ein assoziativer Acht-Wege-Cache ist ein Cachespeicher, in dem pro Adresse bis zu acht Einträge gespeichert werden können [TG01]. Der SRAM befindet sich auf der selben Ebene wie der L2-Cache, ist jedoch selbst nicht cache-fähig [Xil15]. Es gilt bei der Verwendung der cache-kohärenten AXI ACP Schnittstelle zu beachten, dass beim Abrufen und Verändern von zu großen Datenmengen der Prozess des thrashings (Evaluierung nützlicher Daten) zu viel Zeit beansprucht, was zu sehr hohen Latenzen führen kann. Bei größeren Datenmengen sollte der AXI-HP Kommunikationskanal verwendet werden. Jede der vier AXI-HP Schnittstellen ermöglicht einen Datendurchsatz von etwa 1.2 GB/s, was bei vier Kanälen insgesamt 4.8 GB/s entspricht [Xil15].

Kapitel 3

Grundlagen II: Künstliche neuronale Netze

Dieses Kapitel beschreibt die Entwicklung, die Verwendung, den Aufbau und die Funktionsweise von KNN. Neben einer Einleitung und einer Sensibilisierung für die Notwendigkeit und Verwendung von KNN, wird der Aufbau und die Funktionsweise derer nur insofern erläutert, dass die anschließende Implementierung derer auf einem FPGA nachvollzogen werden kann. Weitere Grundlagen und eine ausführlichere Beschreibung künstlicher neuronaler Netze ist in folgenden Werken zu finden: [RW08, KBB+15, Ert08, Kra09]

3.1 Einleitung

Künstliche neuronale Netze (KNN) sind eine, nach dem heutigen Wissensstand, algorithmische Abstraktion der biologischen neuronalen Netze, welche sich in Gehirnen von Lebewesen finden. KNN sind ein Teil des Themas „künstliche Intelligenz". Das Modell eines biologischen Neurons existierte bereits, als Warren McCulloch und Walter Pitts im Jahre 1943 das erste formale Modell eines Neurons erstellten [MP43]. Die Entwicklung der Modelle für KNN nimmt seit je her, besonders stark seit den 1980er Jahren, zu. Einen guten Einblick in die Entwicklung der Theorien zu KNN und dabei entstandenen Modellen gewährt der Sitzungsbericht der Internationalen Workshops zum Thema „Künstliche Neuronale Netze" aus dem Jahr 1995 [MH95]. Die Einsatzszenarien künstlicher neuronaler Netze können in zwei große Themenbereiche eingeteilt werden[RW08]. Einer dieser Themenbereiche deckt die Modellierung von KNN zur Simulation von Gehirnprozessen ab, um das menschliche Verhalten besser zu verstehen zu können. Dieser Themenbereich

wird hier nicht weiter behandelt. Der zweite Themenbereich deckt die Modellierung und Verwendung künstlicher neuronaler Netze, um konkrete Anwendungsprobleme zu lösen, ab. Das KNN, das im Laufe dieser Arbeit auf einem FPGA implementiert wird, ist diesem Themenbereich zuzuordnen. Beispiele solcher Anwendungsprobleme sind die Bild- und Mustererkennung, das Vorhersagen von Ereignissen in der Wirtschaft und die Gerätesteuerung, heutzutage (2016) besonders von Robotern. Eine Definition für künstliche neuronale Netze ist schwierig zu finden, denn ein universales KNN gibt es nicht. Es gibt unterschiedliche Modelle von KNN, die gewisse Prozesse der Signalverarbeitung des Gehirns auf einer Abstraktionsebene möglichst genau nachahmen. Im Grunde können solche Netze als Graphen wahrgenommen werden, die sich je nach Anwendungsproblem, das es zu lösen gilt (genauer der Kategorie des Anwendungsproblems), unterscheiden. All diese Graphen haben jedoch eine Gemeinsamkeit in ihrer Funktionsweise: Sie nehmen Informationen auf, bearbeiten diese, geben die Ergebnisse wieder aus und verändern sich gegebenenfalls, um Fehler auszubessern. Wie das Funktioniert, wird in den anschließenden Kapiteln genauer erläutert. Die Veränderung eines bestehenden KNN ist eine wesentliche Fähigkeit, welche die Netze von festen Algorithmen unterscheidet. Diese Eigenschaft ermöglicht es nämlich, das Lernverhalten von Menschen nachzuahmen. Wichtige Eigenschaften der künstlichen neuronalen Netzes sind die Parallelverarbeitung, die verteilte Speicherung, die (bis zu einem gewissen Maß) Toleranz gegenüber interne Schäden, die Toleranz gegenüber externen Fehlern und die Generalisierung [RW08]. Das biologische neuronale Netz arbeitet vollständig parallel. Die Modelle für KNN bieten ebenfalls einen hohen Grad an Parallelität, müssen jedoch an einigen Stellen, genauer bei jeder Übertragung von einer Bearbeitungseinheit (Neuron) zur nächsten (siehe Abbildung 3.1), synchronisiert werden. Der hohe Grad an Parallelität macht Hardware, die für parallele Berechnungen entwickelt wurde, wie zum Beispiel FPGA oder Grafikkarten, für die Implementierung dieser Netze interessant. Mit verteilter Speicherung ist gemeint, dass die Informationen eines neuronalen Netzes über das gesamte Netz oder große Teile des Netzes gespeichert werden. Die Toleranz gegenüber internen Schäden bewirkt, dass ein kleiner Teil des neuronalen Netzes beschädigt werden kann und die Ergebnisse meist weiterhin ziemlich genau sind. Die Toleranz gegenüber externen Fehlern ist leicht an einem Beispiel zu verstehen. Menschen erkennen ein Auto auch, wenn es durch Hitzeflimmern verzehrt wird. Aktuelle KNN identifizieren ebenfalls verrauschte Bilder, bis zu einem gewissen Maß, ziemlich genau. Die Generalisierung/Kategorisierung der Daten ist ebenfalls mit einem einfachen Beispiel zu erklären. Gegeben sei ein KNN, welches für Bilderkennung verwendet werden kann. Obwohl diesem KNN Bilder mit Objekten aus nur einer Perspektive beigebracht werden, können sie meist die gleichen Objekte aus verschiedenen

Perspektiven erfolgreich erkennen.
KNN sind die erste Wahl, wenn Algorithmen entwickelt werden müssen die sich an Umstände anpassen und sich verändern müssen.

3.2 Aufbau

Ein KNN ist ein gerichteter Graph. Dieser besteht aus Knoten (Neuronen), welche in verschiedenen Schichten angeordnet sind. Die Knoten sind durch gerichtete Kanten (Synapsen) verbunden.

Definition. *Ein Graph ist ein geordnetes Paar $G = (V, E)$, wobei V eine endliche Menge an Knoten und $E \subseteq V \times V$ eine endliche Menge an Kanten bezeichnet. Eine Kante, $e = (u, v) \in E$, verbindet die Knoten $u \in V$ und $v \in V$ miteinander. Ein gerichteter Graph hat zusätzlich die Einschränkung, dass Kanten eine Richtung haben, sodass $(u, v) \neq (v, u), (u, v) \in E, (v, u) \in E$ gilt.*

Zusätzlich sind die Synapsen (Kanten) gewichtet, womit es sich um einen gerichteten und kantengewichteten Graphen handelt.

Definition. *Ein kantengewichteter Graph $G = (V, E, d)$ hat die Eigenschaft, dass jeder Kante $e \in E$ über eine Gewichtsfunktion, $d : E \mapsto \mathbb{R}$, ein Gewicht d_e zugeordnet wird.*

Die Neuronen befinden sich in unterschiedlichen Schichten (Ebenen) und haben je nach Schicht, in der sie sich befinden, unterschiedliche Aufgaben. Die folgende Abbildung stellt ein dreischichtiges KNN dar [KBB+15]:

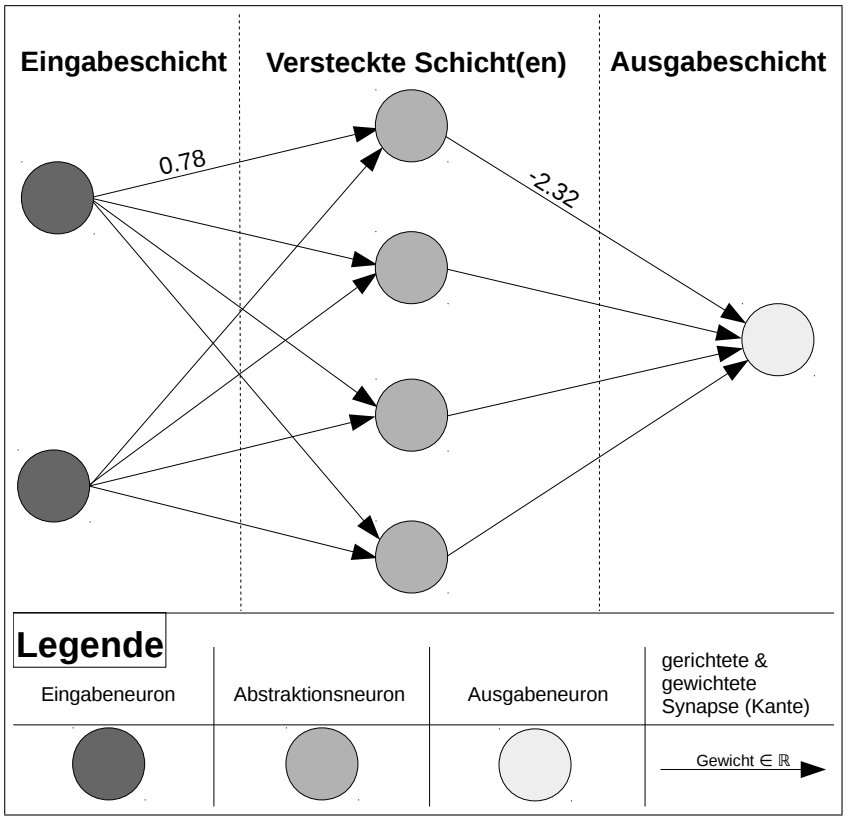

Abbildung 3.1: KNN mit drei Schichten

3.2.1 Knoten (Neuronen) und Kanten (Synapsen)

Neuronen sind Verarbeitungseinheiten, die ein bis mehrere Signale empfangen, diese auswerten und anschließend abhängig von der Auswertung der Signale selbst Signale, über Synapsen, an weitere Neuronen versenden. Je nach dem, in welcher Schicht sich die Neuronen befinden, erfüllen sie verschieden Aufgaben. Es wird grundsätzlich in drei Typen von Schichten unterteilt: Eingabeschicht (Eingabe), versteckte Schicht(en) (Abstraktion) und Ausgabeschicht (Ausgabe) [RW08]. Diese sind im Folgenden beschrieben.

- Neuronen der Eingabeschicht stehen im direkten Kontakt mit der Umgebung. Sie erhalten Signale aus der Umgebung und können als Sensoren aufgefasst werden.
- Neuronen der versteckten Schicht(en) erhalten mindestens ein gewichtetes Eingangssignal und bilden eine Abstraktionsebene. Je mehr versteckte Schichten existieren, umso weiter kann das neuronale Netz abstrahieren.
- Neuronen der Ausgabeschicht erhalten mindestens ein gewichtetes Eingangssignal und befinden sich in der letzten Schicht. Sie geben exakt ein Ausgabesignal als

Ergebnis der gesamten Berechnung aus und können als Aktoren aufgefasst werden. Wie in jeder anderen Schicht können auch in dieser Schicht mehrere Neuronen vorhanden sein.

Die Menge der Eingabeneuronen und die Menge der Ausgabeneuronen muss nicht disjunkt sein, das heißt Eingangsneuronen können gleichzeitig als Ausgabeneuronen agieren, was jedoch nicht die Regel ist [KBB+15]. Die Funktionsweise der Neuronen wird im nächsten Kapitel genauer erläutert.
Neuronen sind über gewichtete und gerichtete Kanten (Synapsen) verbunden. Wenn ein Neuron ein Signal über eine Synapse an ein weiteres Neuron sendet, so wird dieses über die Synapse gewichtet. Eine negative Gewichtung der Synapse führt dazu, dass das empfangende Neuron ein hemmendes Signal empfängt. Eine Gewichtung von Null entspricht dem Nichtvorhandensein dieser Synapse. Eine positive Gewichtung der Synapse hat zur Folge, dass jenes empfangende Neuron vom sendenden Neuron stimuliert wird. [RW08]

3.2.2 Propagierungs-, Aktivierungs- und Ausgabefunktion

Das Neuron eines KNN kann als kleiner Prozessor aufgefasst werden, der ständig Eingangssignale empfängt und drei Funktionen hintereinander durchführt, nämlich die Propagierungsfunktion, die Aktivierungsfunktion und die Ausgabefunktion [KBB+15]:

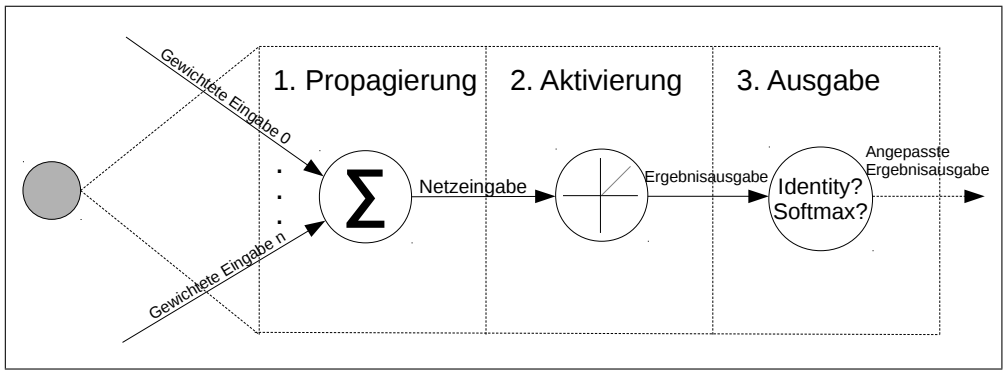

Abbildung 3.2: Darstellung der Funktionen eines Neurons

Die Propagierungsfunktion (auch Netzeingabefunktion genannt) wird zur Bestimmung der Netzeingabe verwendet. Der Begriff der Netzeingabe beschreibt hier den gesamten Einfluss der auf das betroffene Neuron wirkt. Üblicherweise wird für die Propagierungsfunktion die Linearkombination verwendet, welche alle eingehenden Signale mit den entsprechenden Gewichten multipliziert und anschließend kumuliert. [KBB+15]

Für ein Neuron wird die Netzeingabe mittels Linearkombination folgendermaßen berechnet [RW08]:

$$netinput_j = \sum_{i=0}^{N_i} value_{ij} \cdot weight_{ij} \tag{3.1}$$

$netinput_j$ = Netzeingabe des Neuron j in der aktuellen Schicht
i = Nummer des Neurons der vorhergehenden Schicht
j = Nummer des Neurons der aktuellen Schicht
N_i = Anzahl der Neuronen in der i. Schicht (vorhergehende Schicht)
$value_{ij}$ = Wert der von Neuron i an Neuron j übertragen wird
$weight_{ij}$ = Gewicht der Synapse zwischen Neuron i und Neuron j

Nachdem die Netzeingabe berechnet wurde, muss das Neuron anschließend die Signalstärke (das Aktivitätslevel) bestimmen, mit der es Signale an die nächste Schicht, oder über die (letzte) Schicht der Ausgabeneuronen versendet [KBB+15, Ert08]. Die Aktivierungsfunktion bestimmt die Signalstärke der zu sendenden Signale anhand der Netzeingabe. Typische Aktivierungsfunktionen sind die sigmoide Funktion [KBB+15, BFW95],

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

und seit 2011 die biologisch plausiblere Aktivierungsfunktion, Rectifier [GBB11]:

$$f(x) = max(0, x) \tag{3.3}$$

Rectifiers sollen insofern biologisch plausibler sein, da durch die harte Abgrenzung bei Null nur ein Teil der Neuronen im neuronalen Netz gleichzeitig aktiv ist, was in Gehirnen lebender Organismen ebenfalls beobachtet wurde.

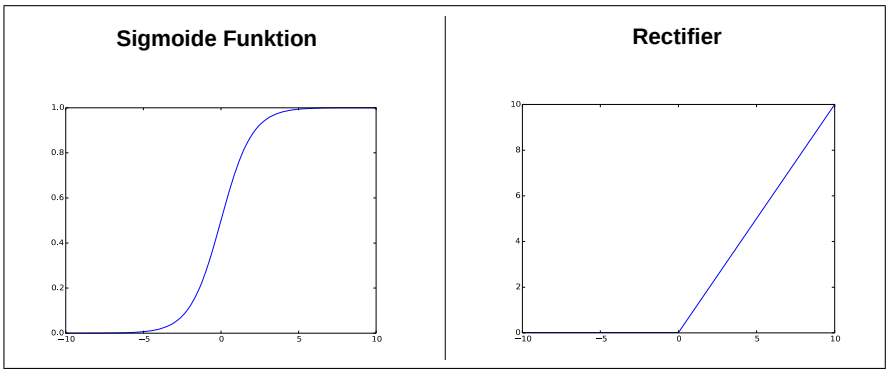

Abbildung 3.3: Links: Sigmoide Funktion, Rechts: Rectifier

Die Ausgabefunktion dient der finalen Anpassung des Signals, bevor das Signal das Neuron verlässt [KBB+15]. Meistens wird hierfür die Identitätsfunktion verwendet. Ein Beispiel einer sinnvollen Ausgabefunktion, abgesehen von der Identitätsfunktion [KBB+15, Ert08], ist die Softmax Funktion [Kra09]. Diese Funktion normalisiert die Werte eines Vektors, sodass der Abstand der Werte im Relation zueinander gleich bleibt, jedoch die Werte summiert Eins ergeben. Der Ergebnisvektor der Softmaxfunktion kann als Vektor aufgefasst werden, der eine Wahrscheinlichkeitsverteilung enthält. Die Softmaxfunktion wird gerne in Neuronen der Ausgabeschicht eines KNN verwendet, welches Klassifizierungen durchführt [KSH12]. Jedes Ausgabeneuron hat hierbei eine Beschriftung, die das erkannte Muster beschreibt. Durch die Verwendung der Softmaxfunktion ergibt sich so eine Wahrscheinlichkeitsverteilung der erkannten Muster.

3.2.3 Training- und Testphase

Künstliche neuronale Netze müssen zunächst trainiert werden, bevor diese die gewünschte Funktion abbilden und generalisieren. Beim Training gilt es zu unterscheiden ob die Topologie des KNN statisch oder dynamisch ist und ob die erwarteten Ergebnisse bekannt sind (supervised learning) oder nicht (unsupervised learning). Es gibt KNN, welche mit einer statischen Topologie initialisiert werden, dass heißt die Anzahl der Neuronen und die Synapsen sind von Beginn an bestimmt [BFW95]. Im Gegenzug zu statischen KNN gibt es auch dynamische KNN, deren Anzahl der Neuronen und Synapsen sich im Laufe des Trainings verändert [Ert08, KBB+15]. Hier wird nur auf statische KNN eingegangen, wobei die erwarteten Ergebnisse der Daten bekannt sind (supervised learning).
Bei statischen KNN werden während der Trainingsphase nur die Gewichte der Synapsen angepasst. Für die Anpassung der Gewichte werden Lernregeln benötigt, die aus den

Ergebnissen des KNN Gewichtsmodifikationen ableiten. Einer der verbreitetsten Trainingsalgorithmen für statische KNN ist Backpropagation [KBB+15, Kra09, Ert08]. Dabei werden Trainingsdaten in das KNN gespeist und das Ergebnis wird mit dem erwarteten Ergebnis abgeglichen. Aus den Unterschieden zwischen erwartetem Ergebnis und tatsächlichem Ergebnis wird der Fehler entnommen und daraus mittels einer Lernregel die Gewichtsanpassung ermittelt [BFW95]. Bei fester Topologie, Propagierungs-, Aktivierungs- und Ausgabefunktionen repräsentieren die Gewichte der Synapsen eines KNN das Gedächtnis des KNN, alle erlernten Informationen sind darin enthalten [RW08].

In der Testphase wird das KNN mit Datensätzen getestet. Hierbei werden die erwarteten Ergebnisse mit den tatsächlichen Ergebnissen verglichen und die Präzision des künstlichen neuronalen Netz wird bestimmt [BFW95]. Neben den Datensätzen die zum Training verwendet wurden sind Datensätze, die nicht zum Training verwendet wurden, interessant, da so bestimmt werden kann wie gut das KNN Daten generalisieren kann [RW08].

3.3 Beispielnetz: XOR

Die „eXclusive OR" (XOR) Funktion mit zwei Eingängen ist eine bijektive Funktion. Das Ergebnis der XOR-Funktion mit zwei Eingängen ist Eins (Wahr), wenn alle Eingabewerte ungleich sind:

x1	x0	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 3.1: Wahrheitswertetabelle der XOR-Funktion mit zwei Eingängen

In dieser Arbeit wird die XOR-Funktion mit zwei Eingängen zum Prüfen der funktionalen Korrektheit der Implementierung des KNN genutzt. Es werden anschließend zwei Varianten der Ergebnisberechnung eines KNN präsentiert.

3.3.1 Netzaufbau

Die XOR-Funktion mit zwei Eingängen lässt sich mit einem Drei-Schichten KNN darstellen, dessen Propagierungsfunktion die Linearkombiniation, die Aktivierungsfunktion „Rectifier" und die Ausgabefunktion die Identitätsfunktion ist:

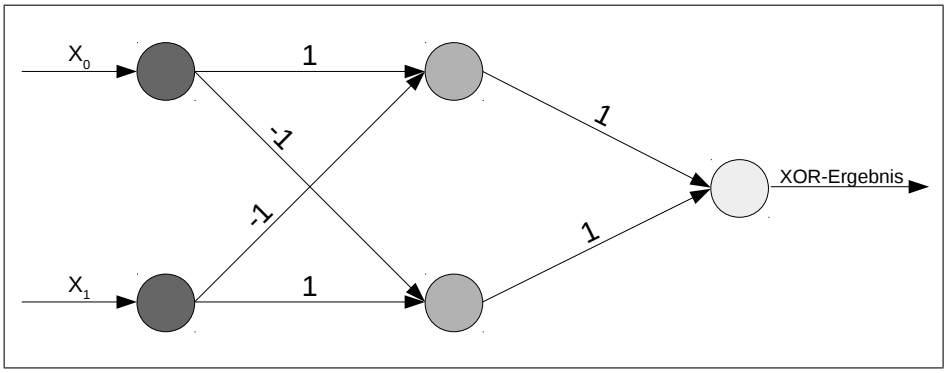

Abbildung 3.4: Drei-Schichten XOR KNN

3.3.2 Ergebnisberechnung durch Pfadabdeckung

Die Ergebnisberechnung durch Pfadabdeckung funktioniert, indem alle Neuronen als eigenständige Prozessoren agieren. Jedes Neuron arbeitet parallel als eigenständige Einheit, wobei die Ausgaben der Neuronen in einer Schicht synchronisiert werden. Jedes Neuron ist selbstständig in der Lage, die Netzeingabe zu berechnen, über die Aktivierungsfunktion das Aktivitätslevel zu bestimmen und über die Ausgabefunktion die Daten über die Synapsen an die Neuronen der nächsten Schicht weiterzuleiten.

3.3.3 Ergebnisberechnung durch Matrixoperationen

Es ist möglich die Netzeingabe aller Neuronen einer Schicht in jeweils einer Matrixoperation zu berechnen. Am einfachsten funktioniert dies für Feedforward KNN, welche Daten nur in eine Richtung, von Eingabe nach Ausgabe, weiterleiten, ohne Schichten zu überspringen. Dafür muss das Skalarprodukt von allen Ausgabewerten der $s-1\ ten$ Schicht mit den Gewichten der $s\ ten$ gebildet werden [RW08]:

$$netinput_s = results_{s-1} \cdot weights_s \tag{3.4}$$

Zunächst folgt eine Definition der Darstellung von Vektoren:

$$\begin{pmatrix} x_0 & \cdots & x_n \end{pmatrix} = \begin{pmatrix} x_0 \\ \vdots \\ x_n \end{pmatrix}^T \in \mathbb{R}^n, n \in \mathbb{N}_0 \tag{3.5}$$

Die Inhalte des Ausgabevektors $results_s$ sind wie folgt definiert:

$$results_s = \begin{pmatrix} x_0 & \cdots & x_n \end{pmatrix} \in \mathbb{R}^n \tag{3.6}$$

x_i = Ausgabewerte des $i\,ten$ Neurons der Schicht s, $i \in [0, .., n] \in \mathbb{N}_0$

Die Inhalte der Gewichtsmatrix $weights_s$ sind wie folgt definiert:

$$weights_s = \begin{pmatrix} w_{00} & \cdots & w_{m0} \\ \vdots & \ddots & \vdots \\ w_{0n} & \cdots & w_{mn} \end{pmatrix} \in \mathbb{R}^{n \cdot m} \tag{3.7}$$

w_{ji} = Gewichte der Synapse zwischen Knoten j der Schicht s und des Knoten i der Schicht $s-1$, $i \in [0, .., n] \in \mathbb{N}_0, j \in [0, .., m] \in \mathbb{N}_0$
Synapsen, die nicht existieren, werden in der Gewichtsmatrix mit dem Gewicht Null festgelegt. Es folgt eine Beispielrechnung, welche eine Eingabe in das XOR-KNN in eine Ausgabe umwandelt:
Gegeben seien die Ausgabewerte der 0. Schicht (1,0), was zugleich die Eingabewerte für die Eingabeneuronen in der 1. Schicht sind:

$$results_0 = \begin{pmatrix} 1 & 0 \end{pmatrix} \tag{3.8}$$

Sowie die Gewichte für die zweite Schicht:

$$weights_2 = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{3.9}$$

Als auch die Gewichte für die dritte Schicht:

$$weights_3 = \begin{pmatrix} 1 & 1 \end{pmatrix} \tag{3.10}$$

Zunächst wird das Ergebnis der ersten Schicht berechnet. Da keine Gewichte für die Ein-

gabewerte vorhanden sind, wird die Aktivitätsfunktion angewandt, um das Aktivitätslevel zu bestimmen ohne zuvor die Netzeingabe berechnen zu müssen. Die Netzeingabe $netinput_1$ entspricht in diesem Fall den Eingabewerten $results_0$. In diesem KNN ergibt das Aktivitätslevel auch die Ausgabe der Schicht, da die Ausgabefunktion in diesem Fall die Identitätsfunktion ist. Die Aktivitätsfunktion Rectifier setzt alle Werte, die kleiner als Null sind, auf Null und belässt die restlichen Werte unverändert:

$$
\begin{aligned}
action\_potential_1 &= \text{Rectifier}\left(netinput_1\right) && (3.11)\\
&= \text{Rectifier}\left(\begin{pmatrix}1 & 0\end{pmatrix}\right) && (3.12)\\
&= \begin{pmatrix}1 & 0\end{pmatrix} && (3.13)\\
&= results_1 && (3.14)
\end{aligned}
$$

Es wird das Skalarprodukt zwischen Ausgabe aus der ersten Schicht und den Gewichten für die zweite Schicht gebildet, was die Netzausgabe für die zweite Schicht als Resultat ergibt:

$$
\begin{aligned}
netinput_2 &= results_1 \cdot weights_2 && (3.15)\\
&= \begin{pmatrix}1 & 0\end{pmatrix} \cdot \begin{pmatrix}1 & -1\\ -1 & 1\end{pmatrix} && (3.16)\\
&= \begin{pmatrix}1 & -1\end{pmatrix} && (3.17)
\end{aligned}
$$

Die Netzeingabe für alle Neuronen der zweiten Schicht wurde berechnet. Anschließend folgt die Berechnung des Aktivitätslevel aller Neuronen der zweiten Schicht und der Ausgabewerte der zweiten Schicht:

$$
\begin{aligned}
action\_potential_2 &= \text{Rectifier}\left(netinput_2\right) && (3.18)\\
&= \text{Rectifier}\left(\begin{pmatrix}1 & -1\end{pmatrix}\right) && (3.19)\\
&= \begin{pmatrix}1 & 0\end{pmatrix} && (3.20)\\
&= results_2 && (3.21)
\end{aligned}
$$

Die gleichen Operationen werden für die dritte Schicht, die Ausgabeschicht durchgeführt:

$$
\begin{aligned}
result &= \text{Rectifier}\left(netinput_3\right) && (3.22)\\
&= \text{Rectifier}\left(result_2 \cdot weights_3\right) && (3.23)\\
&= \text{Rectifier}\left(\begin{pmatrix}1 & 0\end{pmatrix} \cdot \begin{pmatrix}1\\1\end{pmatrix}\right) && (3.24)\\
&= \text{Rectifier}\left(1\right) && (3.25)\\
&= 1 && (3.26)
\end{aligned}
$$

Das Ergebnis (1) stimmt für die Eingabe (0,1).

Kapitel 4

Analyse

In diesem Kapitel werden die grundlegenden Problemstellungen, die es zu bewältigen gilt, erörtert. Sie bilden die Grundlage der Designentscheidungen, die anschließend getroffen werden.
Im Rahmen der verfügbaren Ressourcen auf einem FPGA ist die PL universell programmierbar, da die Sprache in der die Funktionalität der PL beschrieben wird, Turing-vollständig ist. Wird die PL mit einer bestimmten Funktionalität konfiguriert, sind die benötigten Ressourcen festgelegt. Die Funktionalität die abgebildet wird ist ebenfalls bis auf eine Ausnahme fest. Die Ausnahme bildet die Implementierung eines Prozessors auf der PL, da auf diesem durch Turing-vollständige Programmiersprachen unterschiedliche Funktionen sequentiell ausgeführt werden können. Alle parallelisierbaren Operationen werden auf einem FPGA, im Rahmen der verfügbaren Ressourcen, parallel ausgeführt. Die Multiplikation zweier 16-Bit breiter Zahlen kann auf sequentiellen Prozessoren einen Takt dauern. Die Berechnung von zehn Multiplikationen kann in zehn Taktzyklen durchgeführt werden. Auf einem FPGA kann ein Funktionsblock beschrieben werden, der zehn (von einander unabhängige) 16-Bit Multiplikationen gleichzeitig, somit in einem Takt durchführt. Die Berechnung ist auf dem FPGA (bei gleicher Taktfrequenz) somit schneller als auf einem Prozessor, jedoch in diesem Fall auf Multiplikationen festgelegt. Um Additionen durchführen zu können, muss die Funktionalität der PL neu beschrieben werden und anschließend auf dem FPGA konfiguriert werden, was ebenfalls die benötigten Ressourcen verändert.
Die Notwendigkeit, die Funktionalität des FPGA vor der Laufzeit dessen zu definieren, führt zu der Überlegung, welches KNN implementiert wird. Wie am Beispiel der Multiplikation verdeutlicht wurde, muss die abzubildende Funktionalität vor der Implementierung klar sein, um möglichst viele Operationen parallel abzubilden. Alle Eigenschaften, wel-

che die Ergebnisberechnung des KNN verändern, müssen definiert sein bevor das KNN implementiert werden kann. Es handelt sich hierbei um folgende Eigenschaften:

- Anzahl der Schichten
- Richtung des Datenflusses in Synapsen
- Möglichkeit, ob Synapsen Schichten überspringen können
- Funktionen der Neuronen

Eine Änderung dieser Eigenschaften hat zur Folge, dass sich der gesamte Algorithmus zur Ergebnisberechnung und nicht nur einer der Parameter des Algorithmus ändert. Wenn weiterhin das Ergebnis möglichst effizient berechnet werden soll, muss die Funktionalität der PL auf die Änderungen angepasst werden.
Neben diesen Eigenschaften nimmt ebenfalls die Art und Weise der Ergebnisberechnung Einfluss darauf, wie die Funktionalität des KNN implementiert wird. In Abschnitt 3.3 wurden zwei Vorgehensweisen zur Berechnung des Ergebnis eines KNN vorgestellt. Die Implementierung der Pfadabdeckung ist eine exakte Abbildung des neuronalen Netz, wie es beispielsweise in Abbildung 3.1 dargestellt wird. Dabei werden Neuronen als tatsächliche Module implementiert. Die Synapsen werden als Leitungen im FPGA realisiert. Dieses Konzept stellt die dezentrale Implementierung eines KNN dar, dessen Funktionalität in einzelnen Bearbeitungseinheiten über das gesamte neuronale Netz verteilt ist. Jedes Neuron ist eine selbstständige Bearbeitungseinheit, welche die Funktionen eines Neurons alle eigenständig durchführen kann. Die Bearbeitungseinheiten enthalten jeweils alle benötigten Ressourcen in sich, beispielsweise Speicher und DSP, um ihre Berechnungen parallel durchzuführen. Bei der Implementierung nach diesem Modell stellen die Verwaltung der Ressourcen, die Anzahl benötigter Ressourcen (Leitungen,Speicher,DSP) und die Synchronisation der Berechnungen im KNN Schwierigkeiten dar. Die zweite Vorgehensweise beschreibt die Berechnung der Ergebnisse durch Matrixoperationen. Die Neuronen und die Synapsen sind bei einer solchen Implementierung nicht als eigene Einheiten zu identifizieren, sondern die Informationen die sie enthalten, werden in Form von Vektoren und Matrizen abgebildet. Dazu zählen Eingabewerte, Zwischenwerte, Ausgabewerte, und Gewichte und die Zuordnung der Werte zu den entsprechenden Neuronen. Anders als bei der Pfadabdeckung werden hierbei alle Ressourcen zentral verwaltet. Eine zentrale Verwaltungseinheit Steuert die Berechnungen der Ergebnisse für alle Neuronen und beschafft sich über weitere Verwaltungs- und Steuereinheiten die notwendigen Informationen, beispielsweise Gewichte über eine Speicherverwaltungseinheit und das Ergebnis

(festgelegter) arithmetischer Operationen über eine Recheneinheit. Die korrekte Steuerung der Abläufe der Verwaltungseinheit und die Synchronisation der Ergebnisse sind zentrale Probleme dieser Vorgehensweise.
Der Titel dieser Arbeit gibt bereits an, das die Implementierung auf einem FPGA der Zynq-Serie durchgeführt wird. Die Auswahl des exakten Modells muss jedoch noch getroffen werden. Die Auswahl des Modells wird durch das Kriterium bestimmt, dass ein KNN auf diesem (vom Ressourcenverbrauch) implementierbar sein muss und das weiterhin etwas Spielraum besteht, um die Eigenschaften des KNN und die Größe zu verändern. Wichtig bei der Auswahl des Entwickungsboards ist, dass auf dem Entwicklungsboard der gewünschte FPGA verbaut ist und das jenes Entwicklungsboard gut dokumentiert ist und in den verwendeten Tools unterstützt wird.
Die Auswahl einer geeigneten Hardwarebeschreibungssprache/Programmiersprache hängt davon ab, wie die Ergebnisberechnung auf der PL durchgeführt werden soll und wie gut das Design optimiert werden soll. Die Entwicklung eines Hardwaredesigns mit einer Hardwarebeschreibungssprache ist komplex. Anschließende Änderungen am Design erfordern eine lange Einarbeitungszeit. Dafür hat der Entwickler jedoch die Möglichkeit, ein (realistisches) Design nach seinen Vorstellungen zu implementieren, da der Hardwareschaltkreis exakt beschrieben werden kann. Im Vergleich dazu ist es möglich, ein Hardwaredesign in einer höheren Programmiersprache zu entwickeln. Xilinx bietet das Tool SDSoC an, welches semantisch abgewandelten C Quellcode in eine Hardwarebeschreibungssprache und diese anschließend in ein Hardwaredesign umwandelt. Die Entwicklung eines Hardwaredesign mit einer höheren Programmiersprache ist wesentlich einfacher als mit einer Hardwarebeschreibungssprache, da die Entwicklung auf einem höheren Abstraktionslevel durchgeführt werden kann und somit weniger Details beachtet werden müssen. Dadurch werden Entwicklungszeiten kürzer und Änderungen am Design einfacher und schneller durchzuführen. Ein Nachteil ist, dass (realistische) Designs nicht exakt nach den Vorstellungen des Entwicklers implementiert werden können, da die Umwandlung des Quellcodes in eine Hardwarebeschreibungssprache nicht im gesamten Ausmaß kontrolliert werden kann. Viele hardwarespezifische Anweisungen sind über Pragmas steuerbar. Pragmas sind Anweisungen, die nur vom SDSoC Compiler interpretiert und umgesetzt werden. Diese Anweisungen können beispielsweise den Datenaustausch zwischen PS und PL definieren, Berechnungen parallelisieren oder Daten pipelinen. SDSoC eignet sich dazu, ein Hardware/Software Co-Design zu erstellen, in dem die PL als Beschleuniger verwendet wird. Rechenintensive Operationen werden dabei von dem PS auf die PL übertragen, in der durch Parallelisierung und Pipelining die Ergebnisse schneller berechnet werden. Die Berechnung der Ergebnisse eines KNN kann durch Matrixopera-

tionen somit gut in SDSoC programmiert werden. Die Berechnung durch Pfadabdeckung ist ebenfalls in SDSoC implementierbar, eine solche Verwendung von SDSoC liegt jedoch nicht im Sinne des Herstellers und ist sehr aufwändig.
Unabhängig davon, welche Sprache verwendet wird um das Design zu beschreiben, muss vor Beginn der Implementierung definiert werden, welche Datentypen verwendet werden, welche Ressourcen benötigt werden und wie Datenstrukturen abgebildet werden um diese durchführen zu können. Die Ergebnisberechnung findet in den meisten KNN mit 32-Bit Fließkommazahlen (float) statt. Implementierungen von KNN auf sequentiellen Prozessoren verwenden meistens diesen Datentyp. Auf Prozessoren ist eine Hardwareeinheit verbaut, die für schnelle Berechnungen mit Gleitkommazahlen zur Verfügung steht. Diese Hardwareeinheit heißt „Floating Point Unit" (FPU). Da Prozessoren sequentiell arbeiten, kann diese FPU für jede Berechnung nacheinander verwendet werden. Bei FPGA ist das nicht der Fall, wenn Berechnungen parallelisiert werden sollen. Für jede Berechnung mit Gleitkommazahlen, die simultan auf einem FPGA ausgeführt werden soll, muss aus den Logikblöcken (CLB) der PL eine FPU gebaut werden. Die Anzahl der benötigten Ressourcen wird dabei schnell so groß, dass oft die vielen FPU auf der PL des FPGA den Großteil der Ressourcen verbrauchen. Für die Implementierung des KNN auf FPGA eignen sich 32-Bit Gleitkommazahlen somit nicht. Es muss eine alternative Zahlendarstellung gefunden werden, welche die Funktionalität des KNN nicht einschränkt und den Ressourcenverbrauch auf der PL auf ein implementierbares Maß reduziert.
Weiterhin muss ermittelt werden, welche Ressourcen benötigt werden um die Funktionalität eines KNN abbilden zu können. Für die Speicherung von Gewichten, Eingabewerten, Zwischenergebnissen, Ausgabewerten und Konstanten wird Speicher benötigt. Die verfügbaren Speicherbausteine unterscheiden sich auf FPGA von unterschiedlichen Herstellern. Hier werden Speicherbausteine von Xilinx FPGA, speziell der Zynq-7000 Serie, betrachtet. Drei Speicherbausteine wurden bereits in Unterabschnitt 2.1.2 vorgestellt. Der erste Speicherbaustein sind Flip-Flops und Register, die zu Speicherung flüchtiger Daten in kleinen Mengen geeignet sind. Des Weiteren handelt es sich um einen schnellen, in geringer Menge vorhandenen distributed RAM, der aus einem Teil der CLB gebaut wird, welche sich auf der PL befinden. Dieser eignet sich für sehr kleine Mengen an persistenten Daten. Abschließend wurde schneller BRAM Speicher vorgestellt, der sich für das Speichern größerer Datenmengen eignet. Dieser ist über die gesamte PL (an fest definierten Stellen) als tatsächlicher Hardwarebaustein verbaut. Des weiteren ist es möglich, Daten auf DDR2/DDR3/DDR3L oder LPDDR2 Speicher zu speichern, der sich nicht auf dem FPGA, sondern auf der Platine befindet, auf der jener FPGA verbaut wurde. Dieser Speicher hat von allen Speicherbausteinen die größte Kapazität, jedoch zu gleich eine große

Verzögerung zwischen Anfrage und Ergebnis, da die Signallaufzeiten auf Grund der Distanz zur PL groß sind. Zuletzt besteht die Option, Cache-Speicher und SRAM des PS zu verwenden, da die Cache-Kohärenz (Synchronisation der Daten im Cache) zwischen PL und PS aufrecht erhalten wird. Eine effiziente Nutzung dieses Speichers ist auf Grund des thrashings schwierig und die Anwendungsfälle sind speziell. Zusammengefasst ergeben sich folgende Möglichkeiten um Daten zu speichern:

Speicherbaustein	**Signallaufzeit / Latenz**	**Kapazität**	**Datenpfad**	**Menge**
D-Flip-Flops und Register	sehr gering / sehr gering	beliebig	beliebig	gering
distributed RAM	gering / sehr gering	32-256 Bit	32-256 Bit	sehr gering
Block RAM	mittel / gering	36 Kb	72/36 Bit	mittel
DDR2/DDR3 Speicher	groß / groß	beliebig	64 Bit	Ein - Zwei
L2-Cache und SRAM	groß / gering	insg. 768 KB	64 Bit	Einer

Tabelle 4.1: Zynq-7000: Verfügbare Speicherbausteine auf der PL und der Platine

Neben der Notwendigkeit, Daten speichern und abrufen zu können, muss die Möglichkeit bestehen, Daten zu verarbeiten. Arithmetische Operationen können mit CLB berechnet werden. Dabei werden die benötigten Funktionen aus den Logikblöcken gebaut. Alternativ können arithmetische Operationen in festen Hardware DSP Blöcken durchgeführt werden, welche bestimmte Operationen schnell durchführen (siehe Unterabschnitt 2.1.3). Weiterhin muss der logische Ablauf des Schaltkreises definiert werden. Dazu eignen sich letztlich nur CLB. Die Ablage der erforderlichen Informationen, wie beispielsweise der Gewichte, der Anzahl an Neuronen sowie der Zuordnung von Gewichten zu Synapsen erfordert die Wahl geeigneter Datenstrukturen, in der die Daten abgelegt werden können. Die Informationen können als Skalare, als Arrays oder auch als komplexere Strukturen, die sich aus Sklaren, Arrays und weiteren Strukturen zusammensetzen können, gespeichert werden. Die geeignete Wahl von Datenstrukturen und die Definition derer Abbildung in der PL ist erforderlich.
Die Überlegungen darüber, welche Datenstrukturen abgebildet werden und wie Daten auf dem FPGA gespeichert werden können, führt zu der Frage, wie diese Daten auf die PL übertragen werden. Aus dieser Analyse geht bereits hervor, das die PL dazu verwendet wird, die Ergebnisberechnung rechenintensiver Operationen zu beschleunigen. In diesem Fall handelt es sich um die gesamte Ergebnisberechnung eines KNN. Nach der Definition der zu übertragenden Daten und der Analyse derer Struktur, Größe und Verwendung wird ein geeigneter Kommunikationskanal aus den in Unterabschnitt 2.2.3 vorgestellten

Kommunikationskanälen für die Übertragung der Daten vom PS in die PL selektiert.
Der Teil der Implementierung der auf dem Arm Cortex9a Prozessor des PS ausgeführt wird, kann entweder auf einem Betriebssystem oder ohne Betriebssystem (Baremetal) direkt auf der Hardware ausgeführt werden. Je nach Verwendungszweck der Implementierung eignet sich eine der beiden Plattformen besser. Die Programmierung ist auf Betriebssystemen komfortabler, da die Systemressourcen des Computers, beispielsweise der Speicher oder die Ein- und Ausgabeschnittstellen, vom Betriebssystem verwaltet werden. Über vom Betriebssystem bereitgestellte Funktionen, sogenannten „system calls", kann sehr einfach das System gesteuert werden. Der Verwaltungsaufwand, der durch die Verwaltung des Systems über ein Betriebssystem entsteht, hat als Konsequenz, dass die Gesamtleistung des Systems etwas eingeschränkt ist. Soll nur eine Anwendung möglichst schnell ausgeführt werden, ohne Systemressourcen unnötig zu verwalten (unnötig in dem Sinne das sie nicht verwendet werden) und somit Prozessorzeit unnötig zu verbrauchen, bietet sich die Option an, die Implementierung ohne Betriebssystem durchzuführen. Da Systemressourcen nicht automatisch verwaltet werden, entsteht ein zusätzlicher Aufwand, die benötigten Ressourcen selbst zu verwalten.
Ein zentrales Problem der Implementierung stellt die Beschleunigung der Berechnungen dar. Prozessoren berechnen Ergebnisse sequentiell grundsätzlich schneller als FPGA, da diese für sequentielle Abarbeitung von Aufgaben spezialisiert sind und mit wesentlich höheren Taktraten operieren können. Deshalb müssen auf einem FPGA möglichst viele Aufgaben parallel durchgeführt und ein geschickter Datenfluss eingerichtet werden. Es werden anschließend drei Verfahren vorgestellt, welche die Berechnung auf einem FPGA beschleunigen. Das erste Verfahren ist das sogenannte Ausrollen von Berechnungen. Beispiel:

```
1 // 1. Beispiel: Parallelisierung durch ausrollen
2 for (int i = 0; i < 10; i++) {
3     a[i] = b[i] + c[i];
4 }
5 // ...
6 \caption{test}
```

Quellcode 4.1: Sequentieller Schleifendurchlauf

Es ist eine For-Schleife mit zehn Durchläufen zu sehen, in deren Rumpf in jedem Durchlauf ein Array am entsprechenden Index des aktuellen Wertes des Iterators (**`int i`**) durch die Summe der Werte zweier weiterer Arrays an diesem Index beschrieben wird. Da die Schleife sequentiell abgearbeitet wird, benötigt ein Prozessor alle zehn Iterationen um das Ergebnis zu berechnen. Auf einem FPGA, in dem zehn DSP Blöcke zur Verfügung stehen oder in dem aus CLB zehn Volladdierer gebaut werden, können alle Berechnung gleichzeitig ausgeführt werden. Voraussetzung ist hierbei, dass der Speicher, der die Werte für die Variable **`a`** speichert, zehn Schreibzugriffe gleichzeitig durchführen kann und das die Speicher für die Variable **`b`** und **`c`** jeweils zehn Lesezugriffe gleichzeitig durchführen können:

```
1  // 1. Beispiel: Parallelisierung durch ausrollen
2  // Gleichzeitige Ausfuehrung:
3  a[0] = b[0] + c[0];
4  a[1] = b[1] + c[1];
5  a[2] = b[2] + c[2];
6  a[3] = b[3] + c[3];
7  a[4] = b[4] + c[4];
8  a[5] = b[5] + c[5];
9  a[6] = b[6] + c[6];
10 a[7] = b[7] + c[7];
11 a[8] = b[8] + c[8];
12 a[9] = b[9] + c[9];
13 // ...
```

Quellcode 4.2: Paralleler Schleifendurchlauf

Die Parallelisierung der Berechnung funktioniert nicht so einfach, wenn Abhängigkeiten zwischen den Daten bestehen. Für dieses Szenario existiert ein Verfahren, das Datenpipelining genannt wird. Dabei werden über asynchrone Kommunikationskanäle Daten von einer Berechnung in einer andere Berechnung übertragen, sobald diese berechnet wurden. Beispiel:

```
1 // 2. Beispiel: Parallelisierung trotz Abhaengigkeiten mittels Datenpipelining
2 for (int i = 0; i < 10; i++) {
3     a[i] = b[i] + c[i];
4     d[i] = a[i] * a[i];
5 }
```

Die Berechnung kann nicht ausgerollt werden, da **d[i]** von **a[i]** abhängig ist. Die Lösung ist, immer wenn **a[i]** berechnet wurde, das Ergebnis über einen asynchronen Kommunikationskanal an den Funktionsblock weiterzuleiten, der **d[i]** berechnet:

```
1  // 2. Beispiel: Parallelisierung trotz Abhaengigkeiten mittels Datenpipelining
2  for (int i = 0; i < 10; i++) {
3      a[i] = b[i] + c[i];
4  }
5
6  // Zwischen den For-Schleifen existiert ein Kommunikationskanal.
7  // Das Ergebnis von a[i] der ersten For-Schleife wird unmittelbar
8  // an die zweite For-Schleife uebertragen, sobald es berechnet wurde.
9  // Beide For-Schleifen werden so parallel wie moeglich ausgefuehrt.
10
11 for (int i = 0; i < 10; i++) {
12     d[i] = a[i] * a[i];
13 }
```

Quellcode 4.3: Datenpipelining

Schleifendurchläufe können gepipelined werden. Dabei ist nicht das Datenpipelining, sondern die parallele Ausführung der einzelnen Iterationen im Feinen gemeint. In jedem Schleifendurchlauf werden Daten gelesen, möglicherweise Berechnungen durchgeführt und Daten geschrieben. Angenommen es stehen 5 DSP Blöcke zur Verfügung, die Ergebnisberechnung erfordert jedoch die Durchführung von 15 Addition, wie in folgendem Beispiel:

```
1 // 3. Beispiel: Schleifendurchlaeufe pipelinen
2 for (int i = 0; i < 3; i++) {
3     a[i*5]     = b[i*5] + c[i*5]
4     a[i*5 + 1] = b[i*5 + 1] + c[i*5 + 1]
5     a[i*5 + 2] = b[i*5 + 2] + c[i*5 + 2]
6     a[i*5 + 3] = b[i*5 + 3] + c[i*5 + 3]
7     a[i*5 + 4] = b[i*5 + 4] + c[i*5 + 4]
8 }
```

Es werden drei Schleifendurchläufe durchgeführt, in denen jeweils 5 DSP Blöcke zur parallelen Berechnung des Ergebnis verwendet werden. Jeder Schleifendurchlauf kann in folgende Instruktionen unterteilt werden: Daten (aus Speicher) lesen (READ), Berech-

nung durchführen (COMP) und Daten (in Speicher) schreiben (WRITE). Jede dieser Instruktionen wird sequentiell durchgeführt. Folgendes Beispiel zeigt die Instruktionen in jedem Takt der Schleifendurchläufe ohne Pipelining an, wobei die Voraussetzung besteht, dass die Variable **a** in einem Speicher abgelegt ist, der in einem Takt alle Werte schreiben kann. Weiterhin besteht die Voraussetzung, dass die Addition in einem Takt durchführbar ist und die Variablen **b** und **c** in einem Taktzyklus aus dem Speicher gelesen werden können:

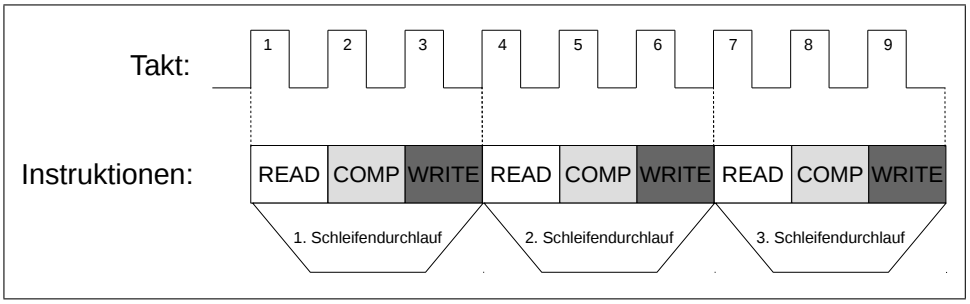

Abbildung 4.1: Schleifendurchläufe ohne Instruktionspipelining

Das Initiation Intervall (II) gibt an, nach wie vielen Taktzyklen der nächste Schleifendurchlauf beginnen soll. Mit II = 1 wird jeden Taktzyklus ein neuer Schleifendurchlauf begonnen. Das Pipelining der Vorschleife sieht mit II=1 folgendermaßen aus:

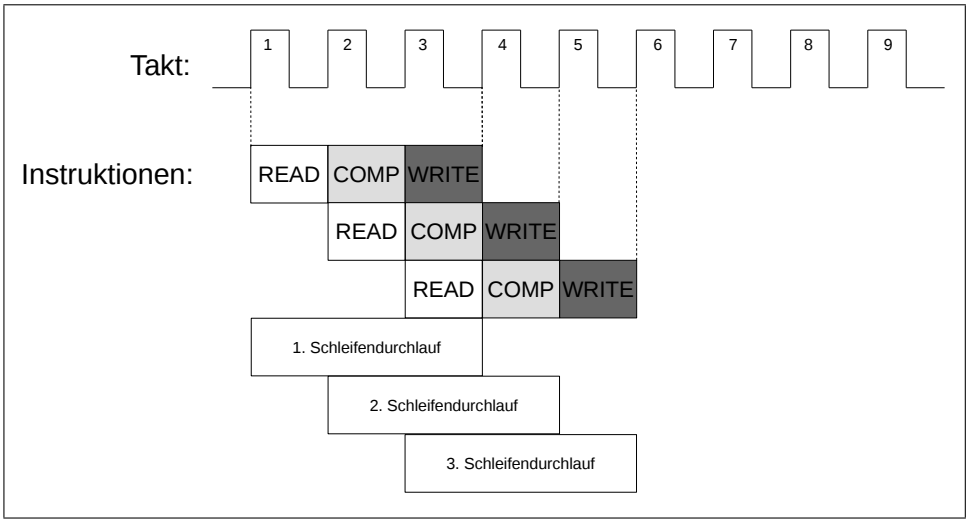

Abbildung 4.2: Schleifendurchläufe mit Instruktionspipelining

Kapitel 5

Design

In diesem Kapitel werden die, aus dem vorhergehenden Kapitel „Analyse" entstandenen, Problemstellungen gelöst. Anschließend werden die Software- und Hardwarekomponenten, die Kommunikation und der funktionale Ablauf in Diagrammen dargestellt. Das Ergebnis dient als Vorlage der Implementierung im nächsten Kapitel.

5.1 Problemstellungen aus der Analyse

Die Lösung der im Kapitel „Analyse" entstandenen Problemstellungen wird in diesem Abschnitt beschrieben. Sie bilden die Grundlage des Layouts, welches im nächsten Abschnitt erstellt wird.

5.1.1 Wahl eines KNN

Das zu implementierende KNN wird drei Schichten haben. Die Entscheidung für drei Schichten wurde getroffen, da mindestens eine versteckte Schicht benötigt wird, um aus der Implementierung ableiten zu können, wie tiefe KNN mit mehreren versteckten Schichten auf FPGA implementiert werden können und welche Ergebnisse bezüglich der Leistung und Skalierbarkeit erwartet werden. Abgesehen von der Auswahl der Anzahl der Schichten gilt die Prämisse, dass die Implementierung möglichst einfach sein soll, jedoch sinnvolle Verwendungen bietet. Diese Prämisse führt zu der Entscheidung, das die Daten nur vorwärts, also von Eingabeknoten über versteckte Knoten zu Ausgabeknoten, verarbeitet werden. Es gibt weder Zyklen, laterale oder rekurrente Verbindungen und es werden keine Schichten übersprungen. Als Propagierungsfunktion wird die meist

verwendete Funktion, die Linearkombination, die sich aus den Eingabewerten und den entsprechenden Gewichten zusammensetzt, ausgewählt. Die Auswahl der Aktivierungsfunktion kann sich je nach Verwendungszweck des KNN unterscheiden, hier gilt jedoch weiterhin die Prämisse, das KNN so einfach wie möglich zu gestalten. Die Sigmoidfunktion $f(x) = \frac{1}{1+e^{-x}}$ ist eine sehr oft verwendete Aktivierungsfunktion. Das Problem bei der Sigmoidfunktion sind rechenintensive Operationen. Außer einem Bruch ist auch eine Exponentialfunktion zu berechnen. Um die Berechnung der Exponentialfunktion auf einem FPGA durchzuführen, werden viele CLB und Zwischenspeicher benötigt. Das würde die maximale Größe des KNN verringern und zu Verzögerungen im Schaltkreis führen. Stattdessen werden Rectifier verwendet $f(x) = max(0, x)$, welche einen geringeren Berechnungsaufwand, einen wesentlich niedrigeren Ressourcenverbrauch und geringere Verzögerungen aufweisen. Die Ausgabefunktion ist die Identitätsfunktion. Sie wirkt somit, als ob sie nicht vorhanden wäre. Die Implementierung einer Ausgabefunktion ist somit nicht notwendig.

Es wird ein KNN mit folgenden Eigenschaften auf dem FPGA implementiert:

- Anzahl der Schichten: Drei
- Richtung des Datenflusses: Nur vorwärts, von Eingabe nach Ausgabe
- Synapsen, die Schichten überspringen: Nein
- Propagierungsfunktion: Linearkombination aus Eingabewerten und entsprechenden Gewichten
- Aktivierungsfunktion: Rectifier
- Ausgabefunktion: Identitätsfunktion

5.1.2 Wahl der Art der Ergebnisberechnung

Die Synchronisation der Ausgaben und die Verwaltung der Ressourcen gelingt beim zentralen Modell (Matrixoperationen) wesentlicher einfacher als beim dezentralen Modell (klassische Vorstellung eines KNN). Beim dezentralen Modell wächst die Anzahl der benötigten Leitungen zwischen allen Neuronen zu schnell mit zunehmender Anzahl an Neuronen. Angenommen es würde das dezentrale Modell auf einem FPGA implementiert werden. Die Zahlen im KNN würden mit 32 Bit dargestellt und das KNN hätte drei Schichten mit jeweils 450 Knoten. Jedes Neuron bräuchte mindestens einen Taktgeber (1

Bit), ein Reset (1 Bit) und die gewichteten Datenleitungen (32 Bit). Die Neuronen einer Schicht wären mit den Neuronen der nächsthöheren Schicht vollständig verbunden, um nicht auf exakt eine Topologie gebunden zu sein. Somit ergäben sich bei einem KNN mit jeweils 450 Neuronen pro Schicht folgende Anzahl an benötigten Leitungen zwischen Neuronen:

$$cost(b, x) = \sum_{i=1}^{N} x_{i-1} \cdot x_i \cdot b \tag{5.1}$$

$$= b \cdot \sum_{i=1}^{N} x_{i-1} \cdot x_i \tag{5.2}$$

$cost(b, x)$ = Erforderliche Leitungen zwischen Neuronen bei klassischer Implementierung, $cost(b, x) \in N^+$
b = Bitbreite, $b > 2, b \in \mathbb{N}^+$
x_i = Anzahl der Knoten der $i\,ten$ Schicht, $x_i \in \mathbb{N}^+$
N = Anzahl der Schichten, $N \in \mathbb{N}^+$
Für 450 Knoten pro Schicht ergäben sich für $cost(34, (450, 450, 450))$ eine Anzahl von 13.770.000 Leitungen, die als Synapsen angesehen werden können. Diese Leitungen würden sich auf Grund der dezentralen Implementierung über die gesamte PL verteilen und die Strecke wird somit sehr groß. Die Ergebnisberechnung durch Pfadabdeckung ist keine Alternative. Die Ergebnisberechnung findet als Reihe von Matrixoperationen statt, wie sie in Unterabschnitt 3.3.3 erklärt wurden.

5.1.3 Wahl des FPGA und des Entwicklungsboard

Für die Implementierung wird der größte FPGA der Zynq-Serie bevorzugt. Das ist der Zynq-7100. Dieser Chip ist nur auf Platinen verbaut, die für bestimmte Anwendungsfälle entwickelt wurden. Diese werden in Xilinx Tools nicht oder nur teilweise unterstützt. Der nächstgrößere FPGA ist der Zynq-7045. Xilinx bietet eine Platine an, auf der dieser verbaut ist. Sie ist sowohl gut Dokumentiert als auch in allen Xilinx Tools vollständig unterstützt. Die Auswahl fällt somit auf den FPGA „Zynq-7045" und das Entwicklungsboard „ZC706" von Xilinx. Die Platine hat zwei 1 GB große DDR3 Speicher verbaut und es ist möglich, über die Netzwerkschnittstelle einen Durchsatz von 1 GB/s zu erreichen. Die folgende Tabelle stellt die für die Implementierung wesentlichen Ressourcen des Zynq-7045 dar [Xil16c]:

CLB	LUT	Flip-Flop	BRAM	DSP
350.000	218.600	437.200	545 · 36 Kb	900

Tabelle 5.1: Zynq-7045: Ressourcen

5.1.4 Wahl der Programmiersprache

Die Entwicklungsumgebung und Programmiersprache wurde bereits in den Zielen dieser Arbeit (Abschnitt 1.3) festgelegt. Die Entscheidung SDSoC zu verwenden und die Entwicklung in C durchzuführen ist primär durch die kürzeren Entwicklungszeiten und die Flexibilität begründet. Es eignet sich für die Beschleunigung intensiver Berechnungen, wie der Ergebnisberechnung eines KNN. SDSoC wird in der Version 2016.1 verwendet.

5.1.5 Definition der Datentypen, Ressourcen und Datenstrukturen

In diesem Abschnitt erfolgt zunächst die Definition der Datentypen. Es werden Datentypen für die Abläufe im KNN und für die Steuerung dessen benötigt. Im KNN werden Gewichte, Ein- und Ausgabewerte sowie Zwischenergebnisse für die Berechnungen benötigt. Wie aus der Analyse bereits hervor geht, ist die Verwendung von 32 Bit Gleitkommazahlen auf Grund des hohen Ressourcenverbrauchs nicht empfehlenswert. In [GAGN15] wurde unter Anderem der Fehler, der beim Training und Test von „deep neural network" (DNN) mit 16 Bit Festkommazahlen entsteht, gemessen (siehe Abbildung 1 im referenzierten Dokument). DNN sind KNN mit mehreren versteckten Schichten. Bei der Messung der Fehler wurden zwei unterschiedliche Rundungsmechanismen verwendet: Zufälliges (stochastisches) Runden und Runden zum nächsten Wert. Dabei hat sich herausgestellt, dass für DNN in beiden Fällen der Fehler vernachlässigbar klein ist, wenn 14 der 16 Bit für die Nachkommastellen verwendet werden. Aus diesem Grund wird für Gewichte, Ein- und Ausgabewerte sowie Zwischenergebnisse folgender Datentyp verwendet:

- Datentyp: 16 Bit Festkomma (fixed point)
- Ganzzahlanteil: 2 Bit
- Nachkommastellen: 14 Bit
- Überlauf: Runden zum größten/kleinsten Wert
- Unterlauf: Runden zum nächsten Wert

Für „Convolutional Neural Networks" (CNN) ist der definierte Datentyp für die Ergebnisberechnung ungeeignet, da der Fehler zu groß wird.
Um das KNN zu steuern, werden Steuersignale benötigt. Dazu gehören Steuersignale zur Bestimmung der durchzuführenden Operation (Initialisierung, Berechnung) und mehrere Steuersignale zur Bestimmung der zur übertragenden Menge an Gewichten und Ein- und Ausgabewerten. Folgende Datentypen werden für die Steuersignale definiert:

- Operationsselektion: unsigned integer 8 bit (Natürliche 8-Bit Zahl)
- Anzahl der Knoten jeweils für jede Schicht: unsigned integer 16 bit

Die zu übertragenden Werte (Gewichte, Ein- und Ausgabe, Steuersignale) werden in möglichst einfache Datentypen verpackt. Dadurch lassen sich die Kommunikationskanäle zwischen PS und PL genauer den zu übertragenden Daten zuordnen. Würden komplexere Strukturen verwendet, müssten Daten aus Strukturen wieder extrahiert werden, um die Daten explizit einem Kommunikationskanal zuzuordnen. Den zu übertragenen Werten werden folgenden Datentypen zugeordnet:

- Gewichte pro Schicht: Eindimensionale Arrays
- Ein- und Ausgabewerte: Eindimensionale Arrays
- Steuersignale: Skalare

Die Ein- und Ausgabewerte des KNN werden vom linken Knoten zum rechten Knoten im Array angeordnet. Das Array, welches die Eingabewerte für das KNN aus Abbildung 3.4 abbildet, enthält beim Index 0 den Wert x0 und bei Index 1 den Wert x1. Die Gewichte sind folgendermaßen angeordnet:
$Gewichte_{(s-1)to(s)} = (w00, w01, w0n, w10, ..., wm0, wm1, wmn)$
$s-1$ = Vorherige Schicht
s = Aktuelle Schicht
$Gewichte_{(s-1)to(s)}$ = Gewichte aller Synapsen zwischen Schicht $s-1$ und Schicht s
m = Knotennummer der Schicht s
n = Knotennummer der Schicht $s-1$
wmn = Gewichte der Synapse zwischen Knoten m der aktuellen Schicht und Knoten n der vorhergehenden Schicht
Abschließend ist für die Erstellung des Designs die Zuweisung von Daten und Aufgaben zu Ressourcen erforderlich. Die Gewichte werden auf Grund der benötigten Kapazität nicht in distributed RAM gespeichert, da dieser nicht in den erforderlichen Mengen

vorhanden ist. Eine Gewichtsmatrix, welche die Gewichte zwischen zwei Schichten mit jeweils 450 Knoten abbildet, hat bereits eine Größe von $450 \cdot 450 \cdot 2$ Bytes (da die Zahlen mit 16 Bit repräsentiert werden). Die Gewichtsmatrix hat in diesem Beispiel eine Größe von 405 KB. BRAM Blöcke haben auf dem Zynq-7045 ausreichend Kapazität, um die Gewichte zu speichern. Aus 545 BRAM Blöcken mit jeweils 36 Kilobit Kapazität ergibt sich eine Gesamtkapazität von 2.452.500 Bytes (2.5 MB). Des Weiteren operieren diese schnell genug und die Signallaufzeiten sind nicht zu hoch. Die Gewichte werden geordnet über BRAM Blöcke verteilt, sodass bei der Ergebnisberechnung des KNN der Durchsatz durch parallele Lesezugriffe möglichst hoch ist. Die Ein- und Ausgabewerte sowie Zwischenergebnisse werden 16-Bit Registern gespeichert. Die benötigte Speicherkapazität dieser Daten ist sehr gering. Der wichtigste Grund für diese Entscheidung ist jedoch, dass die Ein- und Ausgabewerte sowie Zwischenergebnisse jeweils gleichzeitig komplett gelesen und/oder geschrieben werden und ständig schneller Zugriff gewährleistet sein muss, da sie für Berechnungen benötigt werden. Für die Steuersignale reichen 1- bis 16-Bit Register aus.
Auf Grund der Tatsache, das DSP in einem Taktzyklus MACC Operationen durchführen können, werden diese für die Ergebnisberechnung des KNN verwendet. Die Steuerungslogik wird aus CLB gebaut. Folgende Liste fasst die Zuweisungen von Daten und Aufgaben zu Ressourcen zusammen:

- Gewichte: BRAM Blöcke
- Ein- und Ausgabe sowie Zwischenergebnisse: 16 Bit Register
- Steuersignale: 1- bis 16-Bit Register
- Ergebnisberechnung: DSP
- Steuerungslogik: CLB

5.1.6 Zuweisung von Kommunikationskanälen zu Datensätzen

Wie aus dem vorhergehenden Unterabschnitt hervorgeht, werden Gewichte, Ein- und Ausgabewerte sowie Steuersignale zwischen PS und PL übertragen. Die Gewichte sind die Datensätze eines KNN mit dem größten Speicherbedarf. Bei drei Schichten gibt es zwei Gewichtsmatrizen, eine Matrix bildet die Gewichte zwischen der ersten und der zweiten Schicht ab, die Andere bildet die Gewichte zwischen der zweiten und dritten

Schicht. Idealerweise werden die Gewichtsmatrizen beide zu gleichen Zeit, jedoch sequentiell, von PS zu PL übertragen und zur gleichen Zeit in BRAM Speicher geschrieben. Die AXI-GP Kommunikationskanäle eignen sich nur für die Übertragung weniger Bytes, womit diese für die Übertragung der Gewichte ausgeschlossen sind. Beide Matrizen könnten gemeinsam über den einzig vorhandenen ACP Kommunikationskanal übertragen werden, allerdings würde dadurch der Durchsatz sinken. Der Durchsatz würde erstens durch die Reduktion der Kommunikationskanäle, die simultan verwendet werden, und zweitens auf Grund der großen Datenmenge und des dadurch erforderlichen trashings sinken. Unabhängig davon ist die Verwendung des ACP Kommunikationskanal nicht sinnvoll, da Daten nur übertragen und nicht (gemeinsam von PS und PL) bearbeitet werden müssen. Letzteres trifft auf alle Daten zu, die für die Ergebnisberechnung des KNN zwischen PS und PL ausgetauscht werden, weshalb dieser Kommunikationskanal nicht verwendet wird. Auf Grund der Menge der Daten werden die Gewichtsmatrizen über jeweils einem AXI-HP (AFI) Kommunikationskanal übertragen.
Des Weiteren wird ein Kommunikationskanal den Eingabewerten und den Ausgabewerten zugewiesen, welche von dem PS in das KNN auf der PL und von dem KNN in der PL wieder zu dem PS übertragen werden. In einem KNN, in dem die Berechnungen kontinuierlich durchgeführt werden und die Eingaben und Ausgaben dauerhaft übertragen werden (Stream), führt die Verwendung zweier Kommunikationskanäle für Ein- und Ausgabedaten zu einem Leistungsgewinn, da diese Kommunikationskanäle auf Grund des dauerhaften Übertragens von Daten gleichzeitig verwendet werden. Wird kein dauerhafter Datenfluss von der Eingabe über die Berechnung zur Ausgabe realisiert, genügt die Verwendung eines Kanals, da die Übertragungen exklusiv durchgeführt werden. Es wird für Ein- und Ausgabewerte ein gemeinsamer AXI-HP (AFI) Kommunikationskanal verwendet.
Alle Steuersignale werden über den M_AXI_GP Kommunikationskanal übertragen. Hierbei ist, anders als bei allen anderen Kommunikationskanälen, das PS der Master. Das ist insofern sinnvoll, da so das PS die Übertragung initialisiert. Die PL weiß nämlich nicht, wann das PS das KNN initialisieren oder zur Berechnung nutzen will. Die M_AXI_GP Kommunikationskanäle haben jeweils einen eigenen Adressbereich. Führt ein Prozessor eine Schreib- oder Leseoperation auf eine Adresse im Adressbereich der M_AXI_GP Kanäle, werden Daten aus dem Speicher zur PL übertragen und in Registern gespeichert. Die Master und Slave AXI_GP Kanäle verwenden als Übertragungsprotokoll AXI4-Lite, anders als bei allen anderen Kanälen, welche entweder AXI4 oder AXI4-Stream als Protokoll verwenden. AXI4-Lite ist eine Teilmenge von AXI4, weswegen die Implementierung des AXI4-Lite Protokolls weniger Ressourcen benötigt. Dafür ist nicht die gesamte

Funktionalität des AXI4 Protokolls unterstützt, fehlt beispielsweise die Anzahl der Datenübertragungen pro Anfrage. Über das AXI4-Lite Protokoll kann pro Anfrage nur einmal die Menge an Daten übertragen werden, die der Datenbus ermöglicht (ein Burst). Über das AXI4 Protokoll können pro Anfrage bis zu zu 256 mal Daten über den Datenbus übertragen werden (256 Bursts). Für sehr eine kleine Menge an Daten, so wie es bei Steuersignalen der Fall ist, eignet sich der AXI_GP Kommunikationskanal und die Verwendung des AXI4-Lite Protokolls.
Ein dauerhafter Stream an Daten ist für die Gewichte und die Ein- und Ausgabewerte nicht vorgesehen, weshalb für die Übertragung dieser Daten das AXI4 Protokoll (nicht AXI4-Stream) verwendet wird.

5.1.7 Wahl der Plattform

Es bestehen keine Echtzeitanforderungen an die Prozessoren des PS. Lediglich die Übertragung der Daten soll möglichst schnell abgehandelt werden, ohne den Zeitpunkt der Übertragung genau zu definieren. Da keine Echtzeitanforderung an die Prozessoren des PS besteht, kann ein Betriebssystem ohne Echtzeiteigenschaften verwendet werden. Der Vorteil hierbei ist, dass nicht so viele Details bei der Programmierung der Anwendung für die Prozessoren beachtet werden müssen, da bestehende Programmbibliotheken für komplexere Funktionen verwendet werden können, was die Entwicklungszeit verkürzt. Die Erweiterung der Implementierung um Funktionen, wie beispielsweise der eines Servers zur Fernsteuerung des KNN, ist auf Grund der Verwendung eines Betriebssystems vergleichsweise schnell durchgeführt und kann zukünftig entwickelt werden.

5.1.8 Definition der Optimierung

Das Pipelining auf Instruktionsebene (READ, COMP, WRITE) wird für alle Schleifen durchgeführt. Prozessoren sind in dieser Hinsicht wesentlich besser optimiert, weshalb ohne weitere Optimierungen kein Leistungsgewinn erzielt werden kann. Der größte Leistungsgewinn wird aus der parallelen Berechnung der Propagierungsfunktion erreicht. Die Berechnung der Netzeingaben ist im KNN die Berechnung, während der die meisten arithmetischen Operationen in der Testphase durchgeführt werden müssen. Durch die Verwendung von DSP Blöcken werden die Multiplikation und Addition in einem Taktzyklus durchgeführt, was die Bearbeitung eines Eingangswertes (nicht der gesamten Netzeingabe) pro Knoten entspricht.

Die Parallelisierung der Aktivierungsfunktion verspricht den zweithöchsten Leistungsgewinn, da diese nach der Propagierungsfunktion die nächst-meisten Operationen zur Fertigstellung erfordert. Die Aktivierungsfunktion wird in der Implementierung vollständig ausgerollt werden und wird somit komplett parallel ausgeführt. Da Rectifier als Aktivierungsfunktion verwendet wird und keine arithmetischen Ausdrücke enthält, sind keine DSP Blöcke zur Berechnung der Aktivierungsfunktion erforderlich. Für die Berechnung von $f(x) = max(0, x)$ muss lediglich ein Vergleich auf $x < 0$ stattfinden. Das ist besonders einfach, da bei vorzeichenbehafteten Zahlen lediglich das höchstwertigste Bit („Most Significant Bit", MSB) geprüft werden muss um das Vorzeichen zu ermitteln. Es wird ein Ein-Bit-Komparator benötigt um diese Funktion durchzuführen, der aus wenigen CLB gebaut werden kann.

5.2 Layout

In diesem Abschnitt werden alle benötigten Module des Entwurfs modelliert. Dabei müssen viele Komponenten nicht berücksichtigt werden, da SDSoC diese generiert. Dazu zählen u.A. alle AXI Komponenten, Synchronisationskomponenten, Speicherverwaltung und DSP Block Ansteuerung.

5.2.1 Steuereinheit

Die Hardwaresteuereinheit ist das Hauptmodul der Implementierung auf der PL. Sie ist als einziges Modul (nach den AXI Modulen) an alle verwendeten Kommunikationskanäle angebunden. Sie empfängt zunächst Daten bezüglich der durchzuführenden Operation mit den zusätzlichen Informationen, welche Daten übertragen werden und die Menge dieser. Anschließend steuert sie den gesamten Funktionsablauf. Größere Aufgabenblöcke (Speicherverwaltung, Steuerung der DSP Blöcke) werden an weitere Hardwaremodule delegiert. Es folgt eine Zusammenfassung der Aufgaben der Hardwaresteuereinheit:

- Direkte Kommunikation mit PS
- Evaluierung der ausführenden Operation, der zu übertragenen Daten und die Menge dieser
- Ansteuerung von Modulen, dazu zählen:
  - KNN Initialisierungsmodul

- KNN Berechnungsmodul

Folgende Abbildung stellt die Hardwaresteuereinheit mit ihren Verbindungen im Blockdiagramm dar:

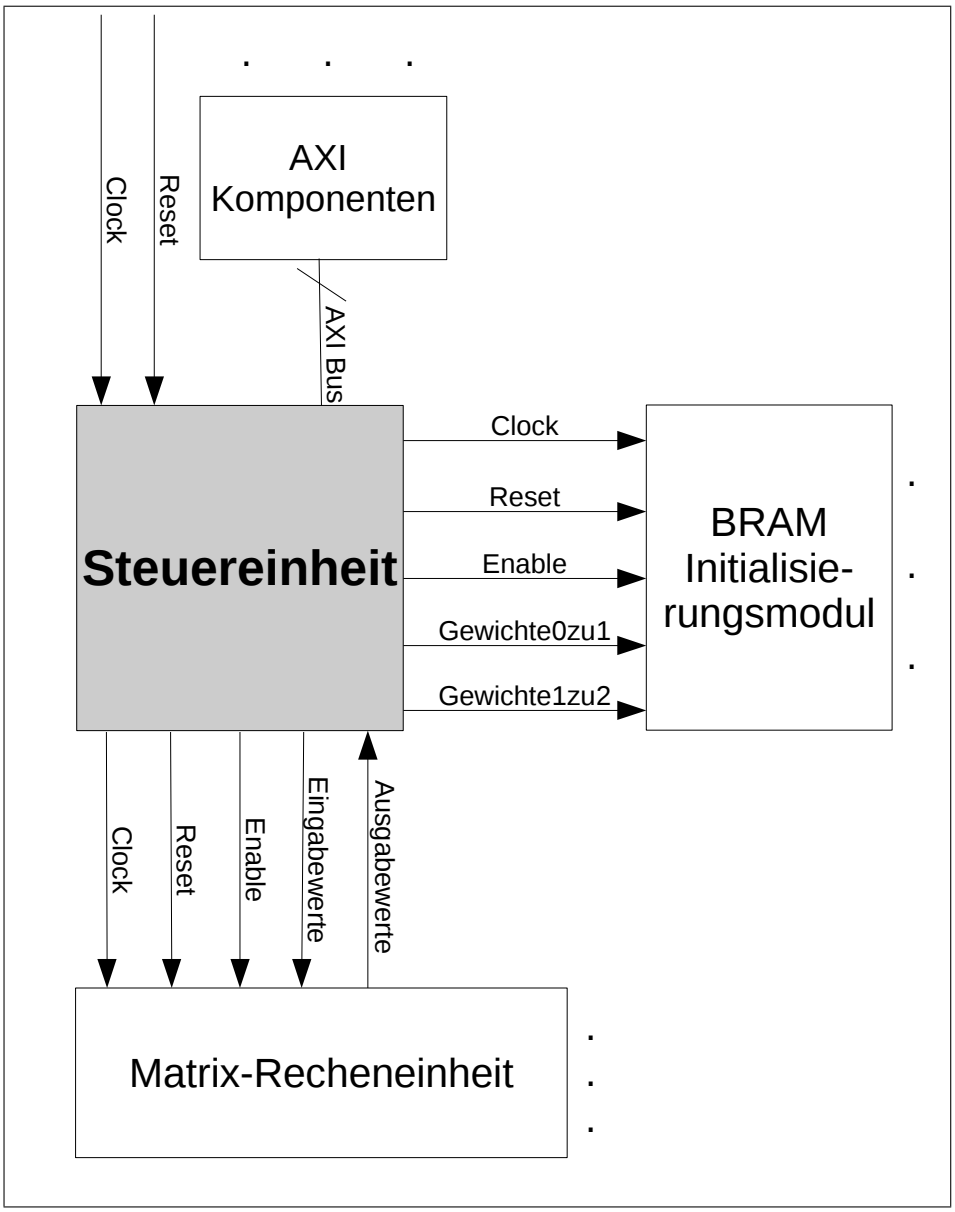

Abbildung 5.1: Darstellung der zentralen Steuerungseinheit auf der PL

5.2.2 BRAM Initialisierungsmodul

Das BRAM Initialisierungsmodul dient der Initialisierung der Gewichte in BRAM Speicher. Für den geordneten Zugriff auf BRAM Speicher wird ein Speicherverwaltungsmodul benötigt. Der BRAM Speicher kann über die Hardwarespeicherverwaltung sinnvoll geordnet, beschrieben und gelesen werden, um den Durchsatz zu maximieren und Konflikte zu vermeiden.

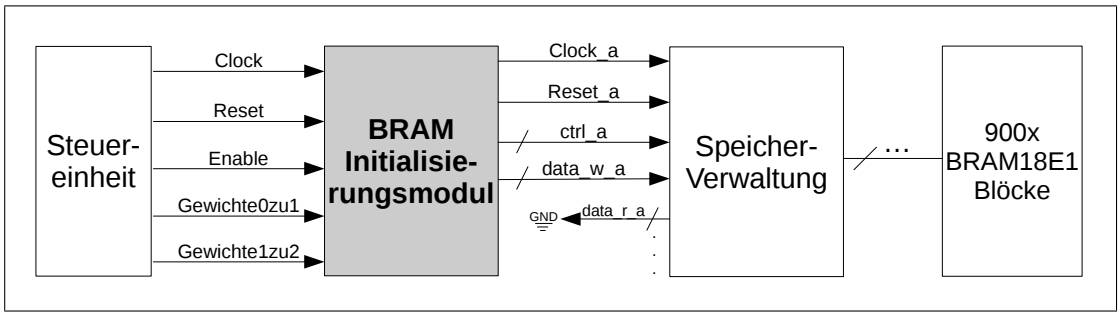

Abbildung 5.2: Darstellung des BRAM Initialisierungsmoduls und der Speicherverwaltung

Die Gewichte werden bei der Initialisierung so auf separate BRAM Blöcke verteilt, dass parallele Lesezugriffe möglich sind und die Matrix-Recheneinheit nicht lange auf die Gewichte aus dem Speicher wartet. Wenn alle, für eine Iteration erforderlichen, Gewichte (z.B. 450 Gewichte) in einem BRAM Block gespeichert sind und der BRAM Block für einen Lesezugriff einen Takt benötigt, wird die Verzögerung der Hälfte der Gewichte (225) an Taktzyklen entsprechen, da auf BRAM Blöcke zwei mal pro Takt zugegriffen werden kann (dual port RAM). Wenn die Gewichte jedoch in 225 (dual port) oder 450 (single port) BRAM Blöcke verteilt sind, können alle Gewichte in einem Taktzyklus abgerufen werden.

5.2.3 Recheneinheit

Die Recheneinheit enthält Eingabewerte und ein Aktivierungssignal von der Steuereinheit. Anschließend berechnet die Recheneinheit die Netzeingabe, das Aktivierungslevel und den Ausgabewerte jedes Neurons, bis die Ausgabewerte der Ausgabeneuronen berechnet wurden und anschließend an die Steuereinheit übertragen werden. Der Algorithmus wird wie in Unterabschnitt 3.3.3 beschrieben in der Recheneinheit implementiert. Dieser stehen 900 DSP Blöcke zur parallelen Berechnung zur Verfügung. Diese Blöcke

werden für die Berechnung der Netzeingabe jeder Schicht verwendet. Dabei werden 450 DSP Blöcke für die Berechnung der Netzeingabe aller Neuronen der versteckten Schicht und 450 DSP Blöcke für die Berechnung der Netzeingabe aller Neuronen der Ausgabeschicht verwendet. Um einen maximalen Durchsatz zu erreichen, werden für alle Neuronen einer Schicht jeweils 450 Teilergebnisse der Netzeingabe parallel berechnet. Dafür werden pro Schicht 450 Register für die Zwischenergebnisse und ein paralleler Zugriff auf 450 Gewichte aus dem BRAM Blöcken benötigt. Es folgt eine Beschreibung des funktionalen Ablaufs:

1. Eingabewerte von Steuereinheit in Register (Eingaberegister) übertragen.

2. Zwischenwerteregister (versteckte Schicht und Ausgabeschicht) parallel mit Null initialisieren.

3. Aktivierungsfunktion (Komparator) parallel auf Eingaberegister durchführen.

4. Netzeingabe der versteckten Schicht (zweite Schicht) aus Eingaberegistern und entsprechenden Gewichten (aus Speicherverwaltung) berechnen (450 MACC Operationen gleichzeitig). Ergebnisse in Register der versteckten Schicht speichern.

5. Aktivierungsfunktion parallel auf Register der versteckten Schicht durchführen.

6. Netzeingabe für Ausgabeschicht berechnen, Ergebnisse in Register der Ausgabeschicht speichern.

7. Aktivierungsfunktion parallel auf Register der Ausgabeschicht durchführen.

8. Ausgaberegister enthalten die Ergebnisse des KNN, Rückübertragung dieser an Steuereinheit.

Abbildung 5.3 zeigt die Einordnung der Recheneinheit in das Blockdiagramm des Designs:

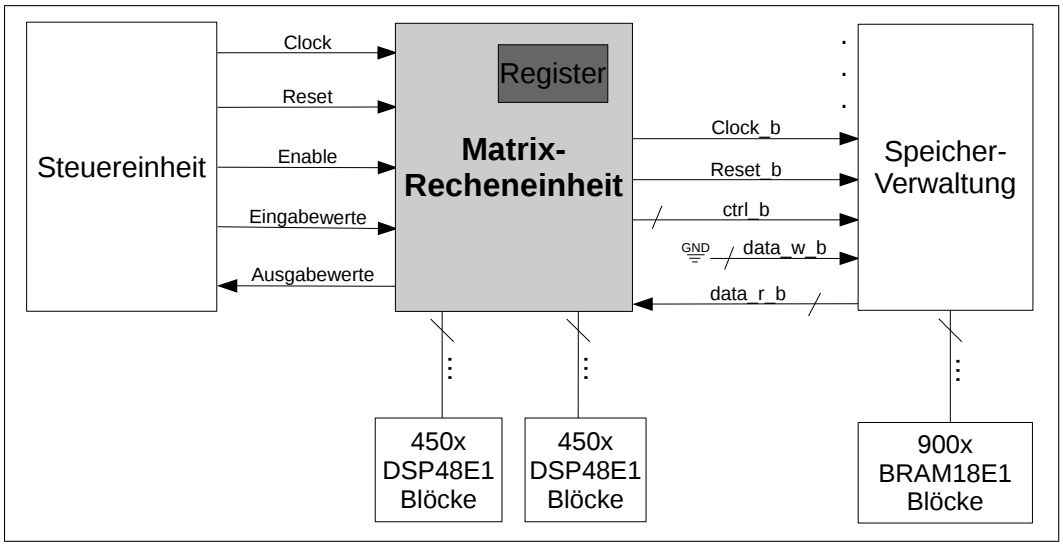

Abbildung 5.3: Darstellung der Recheneinheit auf der PL

5.2.4 Funktionsablauf

Es folgt der Funktionsablauf von der Initialisierung des KNN bis zur Ergebnisausgabe der Hardwareimplementierung:

1. PS initialisiert Gewichte und Eingabewerte in Hauptspeicher.
2. PS führt Ergebnisberechnung des KNN durch und misst die Dauer.
3. PS ruft Hardwarefunktion mit Operation = INIT auf, Speicheradresse der Gewichte und Anzahl der Knoten werden übertragen.
4. PL Steuereinheit ruft BRAM Initialisierungsmodul mit entsprechenden Parametern auf.
5. PL BRAM Initialisierungsmodul holt sich über die Steuereinheit über zwei AXI_HP (AFI) Kanäle Gewichte aus dem DDR3 Speicher.
6. PL BRAM Initialisierungsmodul trägt geordnet Gewichte über die Speicherverwaltung in BRAM Blöcke ein.
7. PS Hardwarefunktionaufruf gibt einen Rückgabewert (Erfolg) zurück.
8. PS ruft Hardwarefunktion mit Operation = EXEC auf, Speicheradresse der Eingabe- und Ausgabewerte sowie Anzahl der Knoten werden übertragen.

9. PL Steuereinheit gibt entsprechende Parameter an Recheneinheit weiter.

10. PL Recheneinheit holt sich über die Steuereinheit über einen AXI_HP (AFI) Kanal die Eingabewerte aus dem DDR3 Speicher.

11. PL Recheneinheit berechnet, wie zuvor in Unterabschnitt 5.2.3 beschrieben, die Ausgabewerte aus.

12. PL Recheneinheit übertragt über die Steuereinheit über einen AXI_HP (AFI) Kanal die Ausgabewerte in den DDR3 Speicher.

13. PS Hardwarefunktionaufruf gibt einen Rückgabewert (erfolg) zurück.

14. PS Vergleicht Ergebnisse und Laufzeiten.

5.2.5 Übersicht

Die Folgende Übersicht der für dieses Projekt benötigten Funktionseinheiten auf dem FPGA schließt das Kapitel Design ab und wird im nächsten Kapitel als Vorlage für die Implementierung verwendet:

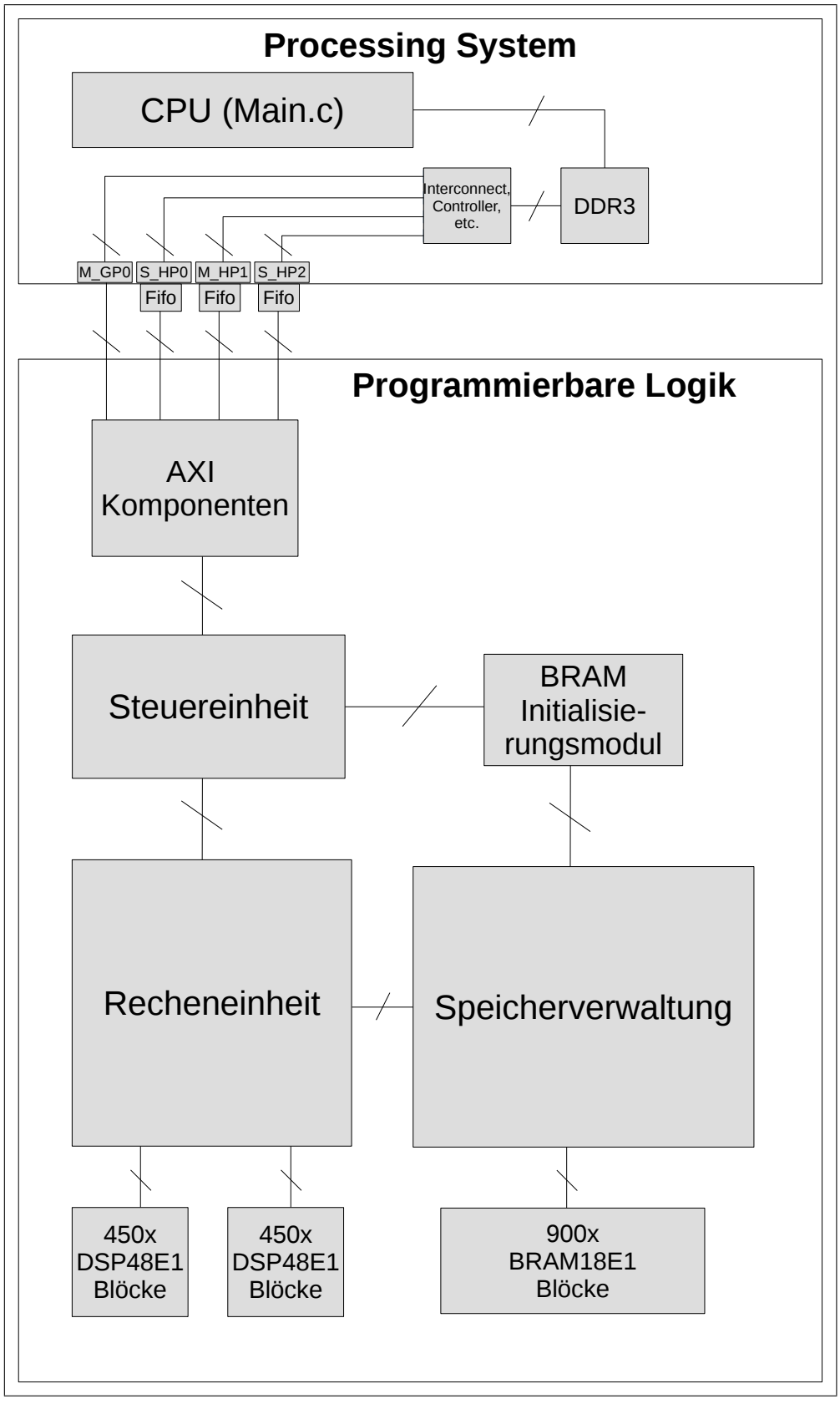

Abbildung 5.4: Übersicht aller für das Projekt benötigten Funktionseinheiten auf dem FPGA

Kapitel 6

Implementierung

6.1 SDSoC

SDSoC ist eine Anwendung, die von Xilinx für Entwicklung von Hardware/Software Co-Designs für Xilinx FPGA der Zynq-Serie in der Sprache C/C++ erstellt wurde. Dabei werden in einem Projekt (in einer Eclipseumgebung oder in der Konsole) C/C++ Anwendungen für den Prozessor und für die programmierbare Logik zugleich entwickelt. Das Resultat der Kompilierung ist ein Abbild, welches von einer SD-Karte in einem FPGA (mit SD-Kartenleser) ausgeführt werden kann.

6.1.1 Quelltextstruktur

Bei der Erstellung einer Quelltextdatei sind alle Klassen und Funktionen dieser Quelltextdatei der Ausführung auf dem Prozessor zugewiesen. Erst die explizite Markierung einer Funktion als Hardware-Funktion lagert diese auf die programmierbare Logik aus. Pro Quelltextdatei kann nur eine Funktion als Hardwarefunktion markiert werden und anschließend von der Softwareapplikation aufgerufen werden. Allerdings können weiter Unterfunktionen erstellt werden, die von der Hardwarefunktion aufgerufen werden. Die Definition der Funktionsprototypen jeder Hardwarefunktion wird von SDSoC in einer separaten Headerdatei gefordert. Dabei können Pragmas angegeben werden, welche beispielsweise Daten einem Kommunikationskanal zuordnen, die zu übertragende Menge und den Offset von Daten festlegen und bestimmen, wie die Daten im Speicher und im Cache verwaltet werden. Pragmas sind Anweisungen für SDSoC Übersetzer und Vivado „High-Level Synthesis” (HLS) Übersetzer, welche die Umsetzung Hardwarespezifischer Anforderungen realisieren.

6.1.2 Funktionsweise

Die Aufgabe von SDSoC ist die Steuerung der Kompilierung von Hardware- und Softwarequelltext, die Erzeugung von notwendigen Metadaten und die Kombination der Resultate zu einem Hardware/Software Co-Design. Dabei ist nicht nur SDSoC involviert, sondern eine ganze Kette von Xilinx Tools, deren Aufrufe SDSoC steuert:

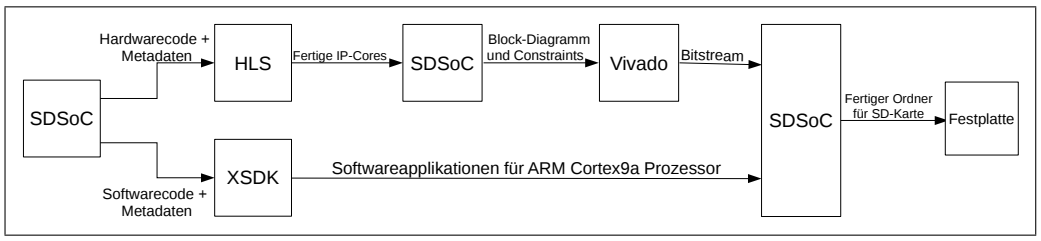

Abbildung 6.1: Darstellung des Datenflusses zwischen Xilinx Tools während der Übersetzung in SDSoC

SDSoC analysiert den Quelltext und sammelt Metadaten. Wenn notwendige Pragmas fehlen (z.B. Definition des Kommunikationskanals für Daten), werden diese Pragmas nach der Analyse des Quelltextes automatisch von SDSoC hergeleitet und angewandt. Die Metadaten werden anschließend mit dem Quelltext, der auf den ARM Cortex9a Prozessor ausgeführt werden soll, an ein weiteres Tool, „Xilinx Software Development Kit" (XSDK), weitergereicht. XSDK erstellt aus den Metadaten und dem Quelltext eine für den ARM Cortex9a ausführbare Datei. Gleichzeitig wird der Quelltext, der auf der PL ausgeführt werden soll, mit den Metadaten an Vivado HLS weitergereicht. Dieser wurde zuvor so von SDSoC angepasst, dass Vivado HLS diesen verarbeiten kann. Bei der Programmierung in C für SDSoC müssen weniger Einschränkungen als bei Vivado HLS beachtet werden, da C darin eine Teilmenge des C in SDSoC ist. Den Quelltext (alle vom Benutzer definierten Hardwarefunktionen) interpretiert Vivado HLS und wandelt diesen in eine Hardwarebeschreibungssprache (VHDL/Verilog) um. Die resultierenden Dateien werden abschließend von Vivado HLS in „Intellectual-Property Cores" (IP-Core) verpackt, ein Format in dem die Funktionsbeschreibung verpackt wird und die Option besteht, Parameter einfach zu konfigurieren. IP-Cores können graphisch in einem Blockdiagramm zusammengeführt werden und ergeben so größere Funktionsblöcke. Anschließend erstellt SDSoC ein Block-Diagramm aus den von Vivado HLS generierten IP-Cores, aus Standard-IP-Cores (z.B. alle AXI-Komponenten) und aus mindestens einer notwendigen Constraintdatei, auf die nicht weiter eingegangen wird. Das Blockdiagramm und die Constraintdatei werden an Vivado weitergereicht. Vivado dient der Interpretierung

von Hardwarebeschreibungssprache und Constraintdateien (beide sind in IP-Cores enthalten), der Optimierung, sowie der anschließenden Abbildung des Ergebnisses auf den gewünschten FPGA. Die resultierende Datei wird Bitstream genannt und wird zur Konfiguration eines FPGA verwendet. Im letzten Schritt werden die Ergebnisse vom XSDK (Softwareapplikation) und Vivado (Bitstream) an SDSoC übertragen und anschließend in einer ausführbaren Struktur verpackt. Der First Stage Bootloader, welcher den Bitstream in die PL lädt und das PS initialisiert, wird automatisch generiert. Bei Bedarf wird ein Linux-Betriebssystem (z.B. Petalinux) integriert. Der Second Stage Boot Loader, der das Betriebssystem initialisiert und der Devicetree, welcher alle notwendigen Informationen der Peripherie enthält, werden ebenfalls generiert.

6.2 Aufbau und Funktionsweise

Das Projekt enthält folgende Quelltextdateien:

- main.cpp - enthält Programmcode für ARM Cortex9a Prozessoren
- neural_net.cpp - enthält die Beschreibung der Hardwarefunktionen (Steuerung, Initialisierung, Berechnung)
- neural_net.h - enthält Definitionen und Informationen zur Hardwarefunktion sowie zum Datenaustausch zwischen dieser und dem PS

Die Anzahl der Knoten und die Verbindungen zwischen diesen (Synapsen) sind im Rahmen oberer Grenzen voll konfigurierbar.

6.2.1 Definitionen, HW-Funktionsprototyp und Datenübertragung

Die Header-Datei der Hardwarefunktion, „neural_net.h”, enthält Konstanten für die durchzuführende Operation oder den Datentyp, der für die Berechnung im KNN verwendet wird. Zusätzlich enthält sie einen Funktionsprototyp der Hardwarefunktion (genauer der Steuereinheit), die von der Softwareapplikation aufgerufen wird. Zuletzt ist die Möglichkeit gegeben, mittels der Angabe von Pragmas Details zur Übertragung der Daten anzugeben.

Definitionen

```
12 #include <stdint.h>
13 #include <ap_fixed.h>
14
15 // define the operation - DO NOT CHANGE - IT WILL REQUIRE A LOT OF ADAPTION
16 #define OPERATION_INIT 0x0
17 #define OPERATION_EXECUTE 0x1
18
19 // define the maximum amount of nodes
20 // in each of the three layers
21 #define MAX_INPUT_NODES 450
22 #define MAX_HIDDEN_NODES 450
23 #define MAX_OUTPUT_NODES 450
24
25 // define fixed-point types
26 #define FP_PRECISION 16
27 #define FP_INTEGER_WIDTH 2
28
29 typedef ap_fixed<FP_PRECISION,FP_INTEGER_WIDTH,AP_RND_CONV,AP_SAT> fixed;
```

Quellcode 6.1: neural_net.h - Zeile 12-29

Das Einbinden der Bibliotheken **`stdint.h`** und **`ap_fixed.h`** dient der Deklaration der erforderlichen arithmetischen Datentypen. **`ap_fixed.h`** ist eine, von Xilinx zur Abbildung von Festkommazahlen entwickelte, Bibliothek. In den Zeilen 16-17 werden Konstanten für die Auswahl der auf dem KNN durchzuführenden Operation definiert. In den Zeilen 21-23 werden jeweils für alle drei Schichten die Maximalanzahl an Knoten definiert. Der Standardwert `450` entspricht der Hälfte der verfügbaren DSP Blöcke. Der Berechnung der Netzeingabe jeder Schicht (versteckte Schicht und Ausgabeschicht) sind jeweils 450 DSP Blöcke zugeordnet. Die Anzahl der DSP Blöcke lässt sich nicht erhöhen, anders als die Definition der maximalen Knoten. Wenn mehr als 450 Knoten für die versteckte Schicht oder die Ausgabeschicht ausgewählt werden, kann die Berechnung der Teilergebnisse aller Knoten der Schicht nicht mehr in einem Durchlauf durchgeführt werden, da nicht genügend DSP (und BRAM) Blöcke vorhanden sind. In den Zeilen 26-29 wird der Festkomma-Datentyp deklariert. Die Deklaration richtet sich nach den in Unterabschnitt 5.1.5 beschriebenen Angaben.

HW-Funktionsprototyp

```
59 void neural_net
60 (\
61         uint8_t op_init,\
62         uint8_t op_exec,\
63         uint16_t inputnodes,\
64         uint16_t hiddennodes,\
65         uint16_t outputnodes,\
66         fixed weights0to1[MAX_INPUT_NODES*MAX_HIDDEN_NODES],\
67         fixed weights1to2[MAX_HIDDEN_NODES*MAX_OUTPUT_NODES],\
68         fixed inputvalues[MAX_INPUT_NODES],\
69         fixed outputvalues[MAX_OUTPUT_NODES]
70 );
```

Quellcode 6.2: neural_net.h - Zeile 59-70

Der vorhergehende Quellcode zeigt den Funktionsprototypen der Steuereinheit, jener Funktion die von der Softwareapplikation aufgerufen wird. Sie enthält zwei Parameter zur Operationsselektion, **`op_init`** und **`op_exec`**. Ein Parameter hätte für diese Aufgabe ausgereicht, jedoch gibt es dabei Komplikationen mit der zu übertragenden Menge an Daten, wie anschließend genauer erläutert wird. Weiterhin gibt es drei Parameter die zur Angabe der Anzahl der Knoten in jeder Schicht dienen, **`inputnodes`** (Eingabeknoten), **`hiddennodes`** (versteckte Knoten) und **`outputnodes`** (Ausgabeknoten). Die letzten vier Parameter nutzen den zuvor deklarierten Festkomma-Datentyp **`fixed`** und dienen der Übergabe der Adresse der Gewichte (**`weights0to1`**, **`weights1to2`**), der Eingabewerte (**`inputvalues`**) und der Ausgabewerte (**`outputvalues`**), sodass die Hardwarefunktion den DDR3-Speicher selbstständig an den relevanten Stellen lesen und schreiben kann. Hierbei ist zu erkennen, dass die maximale Anzahl der Gewichte und Knoten bereits zur Übersetzungszeit (Compile-Zeit) bekannt sein muss. Die genaue Definition der Übertragungsart folgt anschließend.

Datenübertragung

```
35 // transfer data as a stream
36 #pragma SDS data access_pattern(weights0to1:SEQUENTIAL, weights1to2:SEQUENTIAL,\
37         inputvalues:SEQUENTIAL, outputvalues:SEQUENTIAL)
38
39 // define, how many data has to be streamed
40 #pragma SDS data copy(inputvalues[0:inputnodes*op_exec])
41 #pragma SDS data copy(outputvalues[0:outputnodes*op_exec])
42 #pragma SDS data copy(weights0to1[0:inputnodes*hiddennodes*op_init])
43 #pragma SDS data copy(weights1to2[0:hiddennodes*outputnodes*op_init])
```

```
44 // define the memory attributes
45 #pragma SDS data mem_attribute(inputvalues:NON_PHYSICAL_CONTIGUOUS|CACHEABLE)
46 #pragma SDS data mem_attribute(outputvalues:NON_PHYSICAL_CONTIGUOUS|CACHEABLE)
47 #pragma SDS data mem_attribute(weights0to1:NON_PHYSICAL_CONTIGUOUS|CACHEABLE)
48 #pragma SDS data mem_attribute(weights1to2:NON_PHYSICAL_CONTIGUOUS|CACHEABLE)
49 // define data mover
50 #pragma SDS data data_mover(inputvalues:AXIDMA_SG)
51 #pragma SDS data data_mover(outputvalues:AXIDMA_SG)
52 #pragma SDS data data_mover(weights0to1:AXIDMA_SG)
53 #pragma SDS data data_mover(weights1to2:AXIDMA_SG)
54 // define system port
55 #pragma SDS data sys_port(inputvalues:AFI)
56 #pragma SDS data sys_port(outputvalues:AFI)
57 #pragma SDS data sys_port(weights0to1:AFI)
58 #pragma SDS data sys_port(weights1to2:AFI)
```

Quellcode 6.3: neural_net.h - Zeile 35-58

In vorhergehenden Codeabschnitt sind Anweisungen an den SDSoC Übersetzer (Compiler) zu sehen. Es folgt eine kurze Erklärung der Anweisungen:

- **`pragma SDS data access_pattern(arg:type)`**
  Dieses Pragma ermöglicht die Wahl zwischen einem kontinuierlichen Datenfluss zwischen PS und PL, in dem der Entwickler die Daten **`arg`** sequentiell beim Empfang bearbeiten kann, oder einer kompletten Übertragung der Daten in BRAM Blöcke, bevor auf diese zugegriffen werden kann.

- **`pragma SDS data copy(arg[offset:length])`**
  Dieses Pragma bestimmt die Menge der zu übertragenden Daten und den Offset, ab dem Daten übertragen werden. Dabei können Variablen (auch Parameter der Funktion) und einfache arithmetische Operationen (Plus, Minus, Mal, Geteilt) verwendet werden. Der Grund dafür, dass für die Initialisierung des KNN und die Ausführung der Berechnung im KNN zwei Parameter verwendet werden (**`op_init`** und **`op_exec`**), liegt darin, dass keine booleschen Funktion angewandt werden können. Wenn nur ein Parameter die Menge der zu übertragenden Daten bestimmen sollte, ob Gewichte für die Initialisierung übertragen werden oder ob Eingabewerte und Ausgabewerte für die Berechnung übertragen werden, müsste der Parameter in einem Pragma Null ergeben während er im anderen Pragma Eins ergibt und andersherum. Mit den gegeben Mitteln ist dies nicht möglich, weshalb für jede Operation ein separater Parameter angelegt wurde.

- **`pragma SDS data mem_attribute`**
  `(arg[physical_contiguous|cacheable])`

Dieses Pragma bestimmt die Verteilung der Daten des Parameters **arg** im Speicher und ob dieser Parameter auch im Cache gehalten werden kann. Eine Datenstruktur, die im Speicher physisch kontinuierlich abgelegt ist, kann mit weniger Logik wieder abgerufen werden. Genaueres dazu folgt in der Erklärung des nächsten Pragmas. Die Angabe, ob ein Parameter im Cache gehalten wird, nimmt Einfluss darauf ob die Cache-Kohärenz aufrecht erhalten wird und ob vor der Übertragung die Daten des Caches mit den äquivalenten Daten im Speicher synchronisiert werden.

- **pragma SDS data data_mover**(**arg:data_mover**)
  Die Bestimmung eines „data mover" für ein Argument **arg** ist von der Menge der Daten und der Verteilung der Daten im Speicher abhängig. Wenn Daten im Speicher physisch kontinuierlich sind wird keine zusätzliche Logik zur Adressumwandlung benötigt und auf der PL werden weniger CLB verbraucht. Die Standardbibliothek von SDSoC „sds_lib.h" bietet eine Funktion „sds_alloc(...)" an, die zum Reservieren von physisch zusammenhängendem Speichers verwendet werden kann. Die Grenze des so zu reservierenden Speichers liegt bei 8MB. Sind Daten im Speicher nicht physisch kontinuierlich, wird immer zusätzliche Logik benötigt, um diese Daten abrufen zu können, außer es sind weniger als 300 Bytes. Werden variable Datenmengen über das Pragma **SDS data copy**(**...**) für ein Argument **arg** angegeben, wird immer ein „scater-gather direct memory access" Funktionsblock für die Übertragung dieser Daten benötigt.

- **pragma SDS data sys_port**(**arg:port**)
  Dieses Pragma wird zur Selektion des Kommunikationskanals für das Argument **arg** verwendet. Dabei ist lediglich die Auswahl zwischen ACP und AFI gegeben. SDSoC erkennt, dass auf die Gewichte parallel zugegriffen wird, weshalb SDSoC jeder Gewichtsmatrix einen eigenen AFI Kommunikationskanal zuordnet. Des Weiteren erkennt SDSoC, dass die Eingabewerte und die Ausgabewerte nie gleichzeitig übertragen werden, weshalb SDSoC diesen Daten einen gemeinsamen AFI Kommunikationskanal zur exklusiven Nutzung zuordnet.

6.2.2 Steuereinheit

Die Quelltextdatei, die alle Hardwarefunktionen enthält (neural_net.cpp) wird in diesem Unterabschnitt und den folgenden zwei Unterabschnitten erklärt. Die Hauptfunktion dieser Datei ist die, deren Funktionsprototyp bereits in der Quelltextdatei „neural_net.h" deklariert wurde (**neural_net**). Sie wird wie in Unterabschnitt 5.2.1 implementiert. Es

folgt der Funktionsrumpf der Funktion **neural_net**. Die Parameter wurden bereits im vorhergehenden Unterabschnitt erläutert:

```
149     #pragma HLS INLINE self
150     // Define the variables for blockram
151     static fixed bram_weights0to1[MAX_HIDDEN_NODES][MAX_INPUT_NODES];
152     static fixed bram_weights1to2[MAX_OUTPUT_NODES][MAX_HIDDEN_NODES];
153
154   // if the operation is init, call the BRAM initialisation function block
155     if (op_init == 1) {
156         init_neural_net(bram_weights0to1, bram_weights1to2, weights0to1,\
157                         weights1to2, inputnodes, hiddennodes, outputnodes);
158     }
159
160   // if the operation is exec, call the KNN calculation function block
161     if (op_exec == 1) {
162         exec_neural_net(bram_weights0to1, bram_weights1to2, inputvalues,\
163                 outputvalues, inputnodes, hiddennodes, outputnodes);
164     }
```

Quellcode 6.4: neural_net.cpp - Zeile 149-164

Zunächst wird ein Pragma angegeben (**#Pragma HLS inline self**) welches bezweckt, dass die Hardwaremodule auf einer Hierarchieebene implementiert werden. Dies hat den Vorteil, dass eine Optimierung von Vivado über die Grenzen einzelner Module vorgenommen werden kann. Der Hardwareschaltkreis kann somit besser optimiert werden. Anschließend werden die Variablen **bram_weights0to1** und **bram_weights1to2** für den Blockspeicher (BRAM) deklariert. Diese Deklaration hat noch nicht zur Folge, dass Blockspeicher initialisiert wird. Sie verdeutlicht lediglich, dass der Blockspeicher von mehreren Hardwaremodulen verwendet wird. Die Deklaration des Blockspeichers wäre auch im BRAM Initialisierungsmodul möglich und würde funktionieren, dass würde allerdings die Lesbarkeit einschränken. Die Steuereinheit empfängt alle Parameter vom aufrufenden ARM Prozessor, prüft jedoch nur zwei Parameter: **op_init** und **op_exec**. Je nachdem welcher Parameter den Wert Eins enthält, wird die entsprechende Unterfunktion mit den notwendigen Parametern aufgerufen. Es wurde keine Prüfung implementiert, die ausschließt das beide Unterfunktionen aufgerufen werden, weshalb die Hardwarefunktion fehlerhaft verwendet werden kann.

6.2.3 BRAM Initialisierungsmodul

Das BRAM Initialisierungsmodul wird, wie es in Unterabschnitt 5.2.2 beschrieben wurde implementiert. Die dafür Notwendige Speicherverwaltung wird von SDSoC generiert.

Der folgende Quelltext stell den Funktionskopf des BRAM Initialisierungsmodul und die angewendeten Pragmas dar:

```
13 void init_neural_net(fixed bram_weights0to1[MAX_HIDDEN_NODES][MAX_INPUT_NODES],\
14         fixed bram_weights1to2[MAX_OUTPUT_NODES][MAX_HIDDEN_NODES],\
15         fixed weights0to1[MAX_INPUT_NODES*MAX_HIDDEN_NODES],\
16         fixed weights1to2[MAX_HIDDEN_NODES*MAX_OUTPUT_NODES],\
17         uint16_t inputnodes, uint16_t hiddennodes, uint16_t outputnodes)
18 {
19 #pragma HLS INLINE self
20 #pragma HLS dataflow
21 #pragma HLS array_partition variable=bram_weights0to1 cyclic factor=450 dim=1
22 #pragma HLS array_partition variable=bram_weights1to2 cyclic factor=450 dim=1
```

Quellcode 6.5: neural_net.cpp - Zeile 13-22

Die Adresse der BRAM-Speicher **bram_weights0to1** und **bram_weights0to1** wird dem Modul zusammen mit den Datenstream der Gewichte **weights0to1** und **weights1to2** übergeben, damit das BRAM Initialisierungsmodul anschließend den BRAM-Speicher mit den Gewichten initialisieren kann. Um die Menge der zu speichernden Gewichte festzustellen, wird die Anzahl der Knoten jeder Schicht in den Parametern **inputnodes**, **hiddennodes** und **outputnodes** übergeben. Erklärung der Pragmas:

- **pragma HLS dataflow**
  Ohne die Angabe dieses Pragmas würden die Schleifen nacheinander abgearbeitet werden. Mit Angabe dieses Pragmas wird ein asynchroner Kommunikationskanal zwischen alle Schleifen eingefügt, der Daten bei der Berechnung dieser aus einer Schleife in die Andere überträgt. Da zwischen den Schleifen in diesem Hardwaremodul keine Datenabhängigkeiten bestehen, arbeiten beide gleichzeitig, ohne auf Teilergebnisse einer anderen Schleife warten zu müssen. Die Schleifen verhalten sich wie es in Quellcode 4 beschrieben wurde.

- **pragma HLS array_partition variable**=**bram_weights0to1** **cyclic factor**=450 **dim**=1
  Dieses Pragma gibt an, dass der Inhalt des Speichers, der mit der Variable **bram_weights0to1** referenziert wird, auf eine spezielle Art (die gleich genauer erläutert wird) über **factor**=450 BRAM Blöcke verteilt wird. Die Angaben **cyclic** und **dim**=1 gehören zusammen und bedeuten (gemeinsam), dass der Inhalt des Arrays in der ersten Dimension zyklisch auf BRAM Blöcke verteilt wird. Beispiel:

Gegeben Sei ein zweidimensionales Array A = [[1,2,3],[4,5,6],[7,8,9]]. Das Pragma „pragma HLS array_partition variable=A cyclic factor=2 dim=1" wird dem SDSoC Übersetzer die Anweisung geben, das Array folgendermaßen über zwei BRAM Blöcke zu verteilen:

```
1 #pragma HLS array_partition variable=A cyclic factor=2 dim=1
2 int A[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
3 // Inhalt der BRAM Bloecke:
4 BRAM0 = {A[0][0], A[2][0], A[1][1], A[0][2], A[2][2]} = {1,7,5,3,9}
5 BRAM1 = {A[1][0], A[0][1], A[2][1], A[1][2]} = {4,2,8,6}
```

Quellcode 6.6: Beispiel zum Pragma array_partition

Erst wird der Index der ersten Dimension von A hochgezählt, dann wird der Index der zweiten Dimension inkrementiert und anschließend wird wieder der Index der ersten Dimension hochgezählt, bis beide Indizes ihren maximalen Wert erreicht haben. Durch die Verwendung dieses Pragmas wird in der Ergebnisberechnung des KNN die Berechnung von 450 Teilergebnissen gleichzeitig ermöglicht. Es folgt der Rest des Funktionsrumpfes:

```
26     uint16_t i,j,k,l;
27
28     for (j=0; j<MAX_HIDDEN_NODES; j++) {
29         for (i=0; i<MAX_INPUT_NODES; i++) {
30 #pragma HLS PIPELINE II=1
31       // if there is still data to transfer, get it
32       if (i < inputnodes && j < hiddennodes) {
33         bram_weights0to1[j][i] = weights0to1[j*inputnodes+i];
34       }
35       // otherwise initialize the weightmatrix with zeros, so the results of
36       // the caluclation will still be correct
37       else {
38                 bram_weights0to1[j][i] = fixed(0.0);
39             }
40         }
41     }
42
43     for (l=0; l<MAX_OUTPUT_NODES; l++) {
44         for (k=0; k<MAX_HIDDEN_NODES; k++) {
45 #pragma HLS PIPELINE II=1
46             if (k < hiddennodes && l < outputnodes) {
47                 bram_weights1to2[l][k] = weights1to2[l*hiddennodes+k];
48             } else {
49                 bram_weights1to2[l][k] = fixed(0.0);
50             }
51         }
```

```
52      }
```

Quellcode 6.7: neural_net.cpp - Zeile 26-52

Die Schleifendurchläufe werden zum Befüllen der BRAM Speicher verwendet. So wie der Datenfluss in Abbildung 5.2.2 dargestellt wurde, wird er durch die beiden Schleifen realisiert. Die Gewichte werden sequentiell aus dem DDR3 Speicher über jeweils einen AXI_HP (AFI) Kommunikationskanal über die AXI Komponenten auf der PL (Interconnector, AXI_DMA, etc.) in die Steuereinheit übertragen. Diese überträgt die Gewichte an das BRAM Initialisierungsmodul, welches diese, wie in den Pragmas definiert, an die Speicherverwaltung weitergibt. Die Variable **`bram_weights0to1`**, welche direkt auf BRAM Blöcke abgebildet wird, ist folgendermaßen aufgebaut: Der Index der 1. Dimension des Arrays gibt die Nummer des Knoten der versteckten Schicht an, der Index der 2. Dimension des Arrays gibt Nummer des Knoten der Eingabeschicht an. Der Aufruf von $bram\_weights0to1[3][2]$ liefert das Gewicht der Synapse zwischen dem 3. Knoten der versteckten Schicht und dem 2. Knoten der Eingabeschicht. Dies gilt genauso für die Variable **`bram_weights1to2`** für die Gewichte zwischen versteckter Schicht und Ausgabeschicht. Die Gewichtsmatrizen haben die in „neural_net.h" definierten Größen. Werden weniger Gewichte übertragen werden (und das Netz somit weniger Knoten und Kanten hat), so wird der nicht-initialisierte Teil der Gewichtsmatrix mit Nullen befüllt, damit in der anschließenden Berechnung der Ergebnisse keine Fehler entstehen. Das Pragma **`#pragma HLS PIPELINE II`**=1 im Rumpf der verschachtelten Schleife bewirkt, dass die Schleifendurchläufe, so wie es in Abbildung 4 dargestellt wurde, auf Instruktionsebene gepipelined werden.

6.2.4 Recheneinheit

Dieser Unterabschnitt beginnt mit der Erklärung der Parameter der Funktion und der zu Beginn des Funktionsrumpfes definierten Pragmas. Anschließend wird der Funktionsrumpf in drei weitere Abschnitte unterteilt, Initialisierung der Register, Berechnung der Ausgabewerte der versteckten Schicht und Berechnugn und Ausgabe der Ergebniswerte des KNN. Folgender Quelltextausschnitt stellt den Funktionskopf der Recheneinheit und die verwendeten Pragmas dar:

```
55 void exec_neural_net(fixed bram_weights0to1[MAX_HIDDEN_NODES][MAX_INPUT_NODES],\
56         fixed bram_weights1to2[MAX_OUTPUT_NODES][MAX_HIDDEN_NODES],\
57         fixed inputvalues[MAX_INPUT_NODES], fixed outputvalues[MAX_OUTPUT_NODES],\
58         uint16_t inputnodes, uint16_t hiddennodes, uint16_t outputnodes)
59 {
60 #pragma HLS INLINE self
61     static fixed register_hiddenvalues[MAX_HIDDEN_NODES];
62     static fixed register_inputvalues[MAX_INPUT_NODES];
63     static fixed register_outputvalues[MAX_OUTPUT_NODES];
64 #pragma HLS array_partition variable=register_inputvalues complete
65 #pragma HLS array_partition variable=register_hiddenvalues complete
66 #pragma HLS array_partition variable=register_outputvalues complete
```

Quellcode 6.8: neural_net.cpp - Zeile 55-66

Die Adresse der Gewichtsspeicher wird mit den Parametern **bram_weights0to1** und **bram_weights1to2** übergeben. Sie werden für die Berechnung der Netzeingabe jedes Knotens, außer der Knoten der Eingabeschicht, verwendet. Die Parameter **inputvalues** und **outputvalues** sind Datenströme, die aus dem PS über AXI Komponenten und der Steuereinheit an die Recheneinheit angebunden sind (siehe Abbildung 5.4 und Abbildung 5.3). Die Recheneinheit hat die volle Kontrolle darüber, wann die Datenströme gelesen oder geschrieben werden. Die Menge der Knoten jeder Schicht wird mit den Parameter **inputnodes**, **hiddennodes** und **outputnodes** angegeben. Diese Werte werden benötigt um die erforderliche Menge an Eingabewerte und Ausgabewerten mit dem PS auszutauschen und um Berechnungen frühzeitig abzubrechen. Es werden drei Variablen angelegt, welche die Ausgabewerte jeder Schicht enthalten werden: **register_inputvalues**, **register_hiddenvalues** und **register_outputvalues**. Das Pragma **pragma HLS array_partition variable=register_inputvalues complete** gibt an, das die Variable **register_inputvalues** vollständig partitioniert wird, was dazu führt das deren Inhalte in Registern gespeichert werden. Die Register haben durch die Verwendung des Datentyps **fixed** eine Breite von 16-Bit. Die vollständige Verteilung der Elemente der Variable in Registern findet analog für die Variablen **register_hiddenvalues** und **register_outputvalues** statt.

Initialisierung der Register

```
71     uint16_t i,j,k,l;
72
73   // Stream the inputvalues sequentially from the inputstream into registers
74   // and automaticly apply rectifier, so the values for the caclulation of the
```

```
75   // netinput in the next layer are ready
76     for (i=0; i<inputnodes; i++) {
77     #pragma HLS PIPELINE II=2
78         register_inputvalues[i] = inputvalues[i];
79
80     // The following if-branch is the rectifier function
81         if (register_inputvalues[i] < fixed(0.0)) {
82             register_inputvalues[i] = fixed(0.0);
83         }
84     }
```

Quellcode 6.9: neural_net.cpp - Zeile 71-84

Der vorhergehende Codeausschnitt zeigt eine For-Schleife, die dazu verwendet wird die erforderliche Menge an Eingabewerten aus dem DDR3 Speicher im PS abzurufen. Die Werte werden in gleicher Reihenfolge, in der Sie abgerufen werden, in die Register gespeichert. Die Aktivierungsfunktion wird unmittelbar danach auf die Inhalte der Register angewandt, was in den Ausgabewerten des Knoten resultiert. Die Eingabeknoten erhalten jeweils einen Eingabewert, der nicht mehr Gewichtet wird. Sobald die Eingabewerte in die Register übertragen wurden, werden diese die Netzeingabe des Knotens repräsentieren. Da die Ausgabefunktion die Identitätsfunktion abbildet, ist durch die Berechnung des Aktivitätslevels mittels der Aktivitätsfunktion der Ausgabewert des Knotens berechnet worden. Dieser wird für die Berechnung der Netzeingabe der nächsten Schicht verwendet. Anschließend werden die Register für die Netzeingabe, das Aktivierungslevel und die Ausgabewerte der Knoten der versteckten Schicht und der Ausgabeschicht mit Nullen initialisiert.

```
86   // Initialize the registers for the values of the hidden layer fully parallel
87     for (k=0; k<MAX_HIDDEN_NODES; k++) {
88     #pragma HLS unroll
89         register_hiddenvalues[k] = fixed(0.0);
90     }
91
92   // Initialize the registers for the values of the output layer fully parallel
93     for (l=0; l<MAX_OUTPUT_NODES; l++) {
94 #pragma HLS unroll
95         register_outputvalues[l] = fixed(0.0);
96     }
```

Quellcode 6.10: neural_net.cpp - Zeile 86-96

Zunächst werden die Register mit Nullen initialisiert, anschließend wird ihnen die Netzeingabe zugewiesen und schließlich wird die Aktivierungsfunktion auf die Register angewandt, was zugleich die Ausgabewerte eines Knoten ergibt. Die Register werden alle gleichzeitig mit Nullen befüllt. Das Pragma **`pragma HLS unroll`** gibt an, dass

die Schleife wie im Quelltext 4 und im Quelltext 4 dargestellt wurde, parallel ausgeführt wird. es besteht die Möglichkeit, einen Faktor anzugeben, der bestimmt wie viele Iterationen der Schleife ausgerollt werden.

Berechnung der Ausgabewerte der versteckten Schicht

Der folgende Codeausschnitt repräsentiert die Berechnung der Netzeingabe der versteckten Schicht:

```
98   // Calculation of the netinput for the hidden layer
99     for (i=0; i<MAX_INPUT_NODES; i++) {
100        if (i > inputnodes) break;
101        for (j=0; j<MAX_HIDDEN_NODES; j++) {
102 #pragma HLS unroll factor=450
103            fixed product = register_inputvalues[i] * bram_weights0to1[j][i];
104            register_hiddenvalues[j] += product;
105        }
106    }
```

Quellcode 6.11: neural_net.cpp - Zeile 98-106

Für alle Knoten der versteckten Schicht wird zunächst der Ausgabewert des ersten Knotens der Eingabeschicht und dessen Gewicht multipliziert, und anschließend dem entsprechenden Ergebnisregister der versteckten Schicht aufsummiert. Die folgende Abbildung stellt die Beschriebene Berechnung dar:

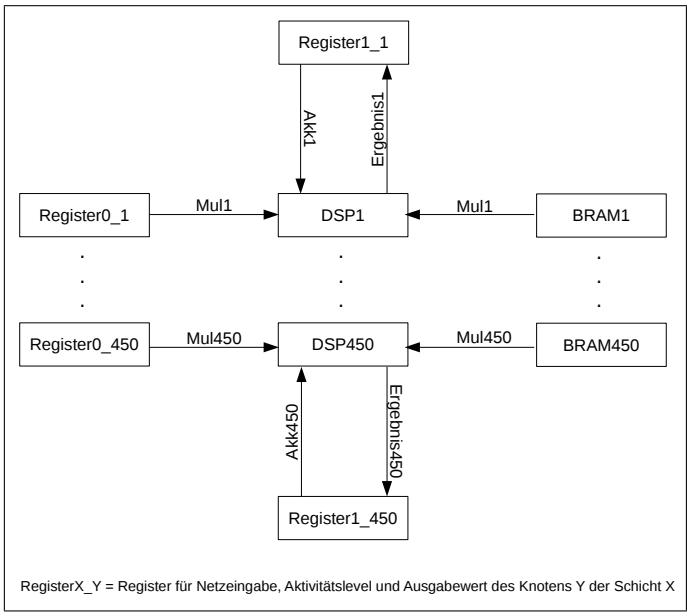

Abbildung 6.2: Darstellung der Matrix-Recheneinheit

Das Gleiche wird für alle weiteren Knoten der Eingabeschicht durchgeführt. Anschließend ist die Netzeingabe aller Knoten der versteckten Schicht in den Registern der versteckten Schicht gespeichert. Die innere Schleife kann nicht um einen dynamischen Wert ausgerollt werden. Würde in der inneren Schleife ein frühzeitiger Abbruch in Abhängigkeit einer Variable stattfinden (in dem Fall **`hiddennodes`**), so wird der SDSoC Übersetzer während der Übersetzung keine Kenntnis darüber haben, wie viele DSP Blöcke benötigt werden und als Konsequenz die Schleife (trotz des Pragmas) sequentiell ausführen. Wenn das Pragma **`pragma HLS unroll factor`**=`450` ohne die Einschränkung **`factor`**=`450` angegeben wird, kann das Design ab einer bestimmten Anzahl an Knoten auf dem FPGA auf Grund einer zu geringen Anzahl an vorhandenen DSP Blöcken und CLB nicht implementiert werden. Der Faktor gibt hier somit eine obere Grenze der zu verwendenden DSP Blöcke und somit des Grades an Parallelität an. Nach der Berechnung der Netzeingabe wird die Rectifierfunktion vollständig parallel auf alle Ergebnisregister der versteckten Schicht angewandt:

```
108     // Apply Rectifier fully parallel
109     for (i=0; i<MAX_HIDDEN_NODES; i++) {
110 #pragma HLS unroll
111         if (register_hiddenvalues[i] < fixed(0.0)) {
112             register_hiddenvalues[i] = fixed(0.0);
113         }
114     }
```

Quellcode 6.12: neural_net.cpp - Zeile 108-114

Berechnung und Ausgabe der Ergebniswerte des KNN

Der anschließende Quelltext dient der Berechnung der Netzeingabe aller Knoten der Ausgabeschicht. Die Funktionsweise der folgenden For-Schleife ist der Funktionsweise der For-Schleife identisch, welche der Berechnung der Netzeingabe der versteckten Schicht dient. Die Anzahl der Iterationen sowie der Gewichte und der Register, die zur Ergebnisberechnung verwendet werden, unterscheiden sich in Abhängigkeit der Anzahl der Knoten der Schicht:

```
116   // At this point, we have got the outputvalues for the hiddenlayer
117   // Calculation of the netinput for the output layer
118     for (i=0; i<MAX_HIDDEN_NODES; i++) {
119         if (i > hiddennodes) break;
120         for (j=0; j<MAX_OUTPUT_NODES; j++) {
121 #pragma HLS unroll factor=450
122             fixed product2 = register_hiddenvalues[i] * bram_weights1to2[j][i];
```

```
123             register_outputvalues[j] += product2;
124         }
125     }
```

Quellcode 6.13: neural_net.cpp - Zeile 116-125

Nachdem die Netzeingabe aller Ausgabeneuronen berechnet wurde, wird die Aktivierungsfunktion sequentiell auf alle Register der Ausgabeschicht ausgeführt. Dabei wird unmittelbar, nach der Anwendung der Aktivierungsfunktion auf ein Ausgaberegister, das Ergebnis zurück zum PS übertragen. Die Berechnung der Ausgabewerte und die Übertragung der Ergebnisse finden (mit einer minimalen zeitlichen Verschiebung) zur gleichen Zeit statt:

```
127     // Apply rectifier sequential this time and transfer each output value
128   // immediately to the PS when it is calculated, so the calculation and the
129   // transfer of the caluclated data happen in parallel
130     for (i=0; i<outputnodes; i++) {
131 #pragma HLS PIPELINE II=1
132         outputvalues[i] = ((register_outputvalues[i] < fixed(0.0)) ? fixed(0.0) :
                register_outputvalues[i]);
133     }
```

Quellcode 6.14: neural_net.cpp - Zeile 127-133

Kapitel 7

Auswertung

Der Datendurchsatz der Implementierung sowie der Verbrauch des FPGA werden während der Berechnung gemessen und mit weiteren Plattformen verglichen. Anschließend wird auf die Übersetzungszeit des Hardware/Software Co-Designs und die verbrauchten Ressourcen eingegangen. Darauf folgen Vorschläge für die Optimierung des Durchsatz der Implementierung und eine Bewertung von SDSoC, sowie abschließend ein Fazit.

7.1 Benchmark

Zur Verifikation der funktionalen Korrektheit der Implementierung wurde ein KNN verwendet, welches die XOR Funktion mit zwei Eingängen abbildet. Dieses lieferte für alle gültigen Eingaben das richtige Ergebnis. Um die Performanz der Implementierung zu bestimmen, wird ein zufälliges KNN initialisiert und mit zufälligen Eingabewerten getestet. Die Laufzeit wird durch den Arm Prozessor auf dem PS gemessen. Die Implementierung wird ebenfalls auf dem Prozessor ausgeführt. Der Datentyp der zur Berechnung in der PL verwendet wird unterscheidet sich vom Datentyp der zur Berechnung auf dem Prozessor verwendet wird. Auf dem Prozessor werden 32-Bit Fließkommazahlen (float) zur Berechnung verwendet, da die Berechnung wesentlicher schneller das Ergebnis bestimmt als mit Festkommazahlen (16-Bit fixed). Das liegt daran, dass die Berechnung von Festkommazahlen durch keine Hardwareeinheit beschleunigt wird und viele Prozessorinstruktionen ausgeführt werden. Da der Prozessor sequentiell arbeitet und somit eine Berechnung nach der anderen durchführt, ist die Verwendung der Hardware-FPU zur Berechnung von Fließkommazahlen wesentlich effizienter als die Verwendung von Prozessorinstruktion zur Berechnung von Festkommazahlen.

Die Implementierung auf dem FPGA kann mit einer Frequenz von 100,00 MHz betrieben werden. Der nächsthöhere vom PS übermittelte Takt beträgt 142,86 MHz und ist bereits für die korrekte Funktionsweise des Hardwareschaltkreises zu hochfrequent. In folgender Abbildung ist ein Leistungsvergleich zwischen einem Arm Cortex9a Prozessor und einem Zynq-7045 zu sehen. Beide Systeme verwenden die gleiche, für das jeweilige System angepasste Implementierung. Die Anzahl der Knoten ist in jeder Schicht gleich und bewegt sich im Rahmen $Anzahl \in [1, .., 50] \in \mathbb{N}^+$:

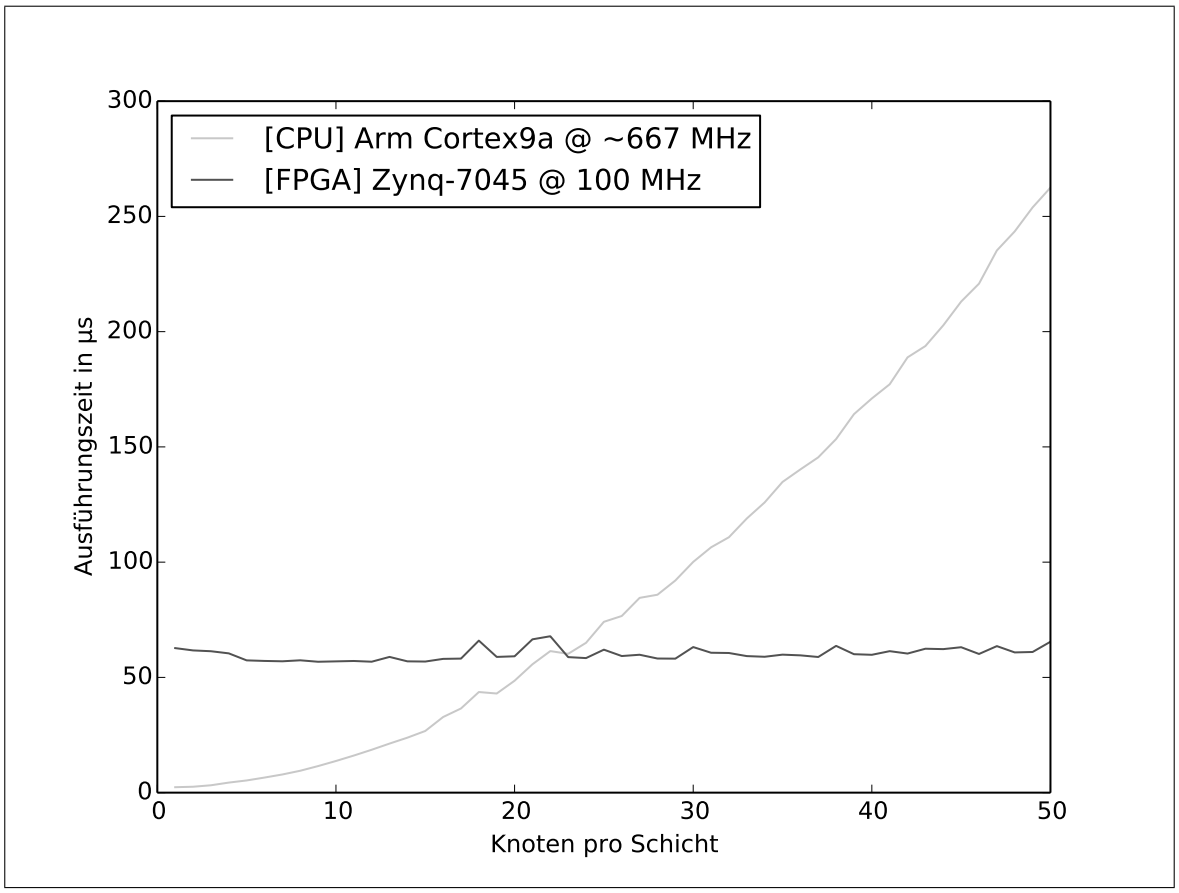

Abbildung 7.1: Geschwindigkeitsvergleich: Berechnung des Ergebnis des KNN, 1 bis 50 Knoten pro Schicht

Die gleiche Darstellung für $Anzahl \in [1, .., 450] \in \mathbb{N}^+$:

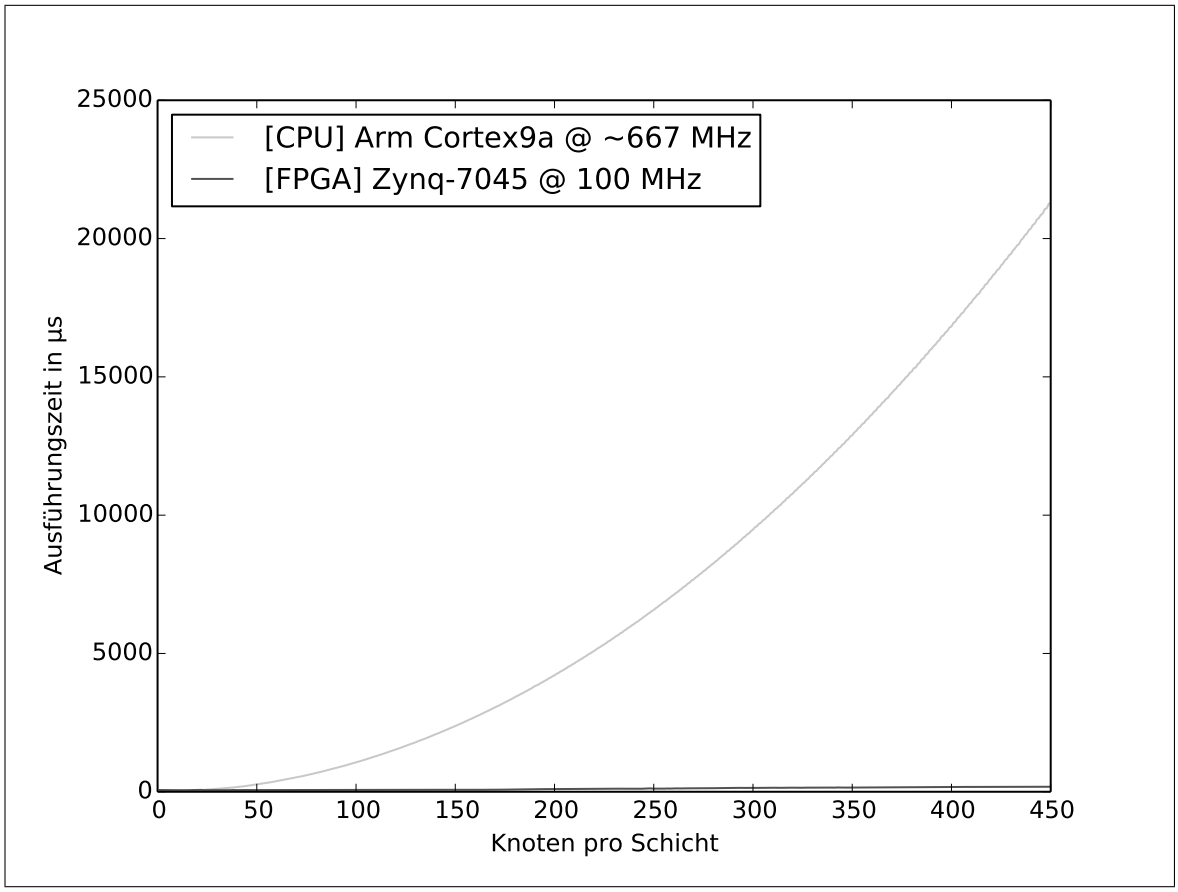

Abbildung 7.2: Geschwindigkeitsvergleich: Berechnung des Ergebnis des KNN, 1 bis 450 Knoten pro Schicht

Es ist zu erkennen, dass ab etwa 23 Knoten pro Schicht der FPGA schneller ist als der Prozessor. Weiterhin ist zu erkennen, dass mit zunehmender Anzahl an Knoten pro Schicht der Geschwindigkeitsgewinn durch die Auslagerung der Berechnung auf den FPGA im Vergleich zum Prozessor stark zunimmt.
Durch die Verwendung des Analog-zu-Digital Konverters auf dem FPGA kann die elektrische Leistung gemessen werden. Der auf dem FPGA vorhandene Analog-zu-Digital Konverter heißt XADC. Für die Verwendung des XADC in SDSoC muss vorher in einer Hardwarebeschreibungssprache oder mit IP-Cores in einem Blockdiagramm ein Hardwaredesign erstellt werden und als Plattform in SDSoC integriert werden. Dieses Vorgehen wurde nicht angewandt um die elektrische Leistung zu messen. Stattdessen wurde die elektrische Leistung durch den Spannungsabfall an einem 1 Ohm Messwiederstand hergeleitet. Die folgende Aufnahme zeigt die Messung der Spannungsabweichung an einem Digitaloszilloskop:

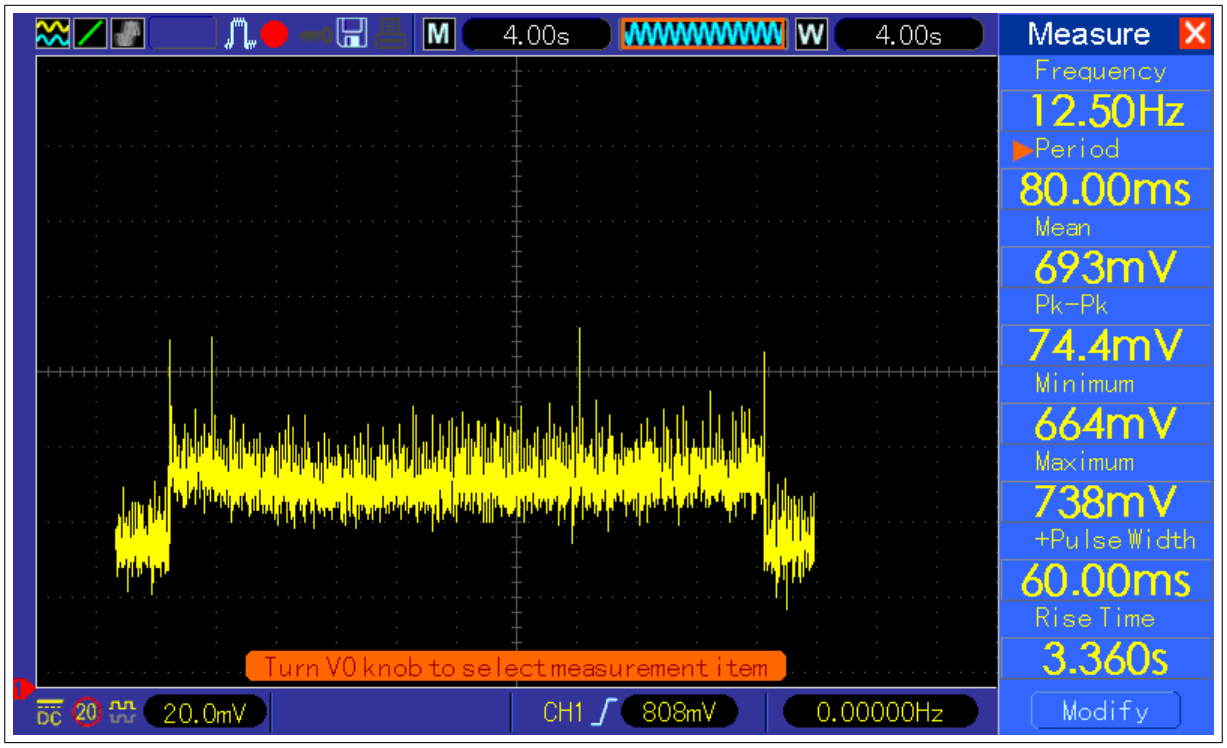

Abbildung 7.3: Aufnahme der Spannungsmessung am Digitaloszilloskop

Der Stromversorger zeigt die aktuelle Stromstärke an. Aus den Messungen ergibt sich ein elektrischer Grundverbrauch der gesamten Entwicklungsplatine von etwa 6,9744 Watt und während der Berechnung ein zusätzlicher Verbrauch von etwa 0,228 Watt. Der Gesamtverbrauch während der Berechnung beträgt etwa 7,2024 Watt. Eine Nvidia Tesla P40 GPU (2016) hat im Vergleich einen Gesamtverbrauch von bis zu 250 Watt [Nvi16].
Während der Übersetzung des Hardware/Software Co-Designs in SDSoC kann unter Umständen viel Speicher benötigt werden. Der Höchstwert beträgt während der Übersetzung ca. 24 GB. Vivado HLS kann nur auf einem Prozessorkern ausgeführt werden. Üblicherweise nimmt die Übersetzungszeit in Vivado HLS die meiste Zeit der Übersetzung in SDSoC ein. In diesem Projekt sind es ungefähr 75%. Die restlichen 25% der Übersetzung können hoch parallel ausgeführt werden, sofern genügend Prozessoren / Prozessorkerne verfügbar sind. Die Übersetzung auf einem Server bietet sich auf Grund der Menge des vorhandenen Arbeitsspeichers, der von Vivado HLS benötigt werden kann, und auf Grund der hohen parallelen Arbeitsweise von Vivado, an. Der endgültige Ressourcenverbrauch des Hardwaredesigns ist in folgender Tabelle abgebildet:

Baustein	Vorhanden	Verbraucht	Verbraucht%
CLB	54650	31665	57.94
CLB (LUT)	218600	83838	38.35
CLB (Flip-Flops)	437200	77149	17.65
DSP48E1	900	900	100.00
BRAM18E1	1090	914	83.85

Tabelle 7.1: Zynq-7045: Ressourcenverbrauch der Implementierung

7.2 Problemstellen

In der Implementierung dieser Arbeit wurden Daten nicht gepipelined. Die parallele Berechnung der Ergebnisse zweier Schichten durch Austausch berechneter Daten findet nicht statt. Das Problem entsteht aus folgenden Grund: In dem in dieser Arbeit angefertigten Design werden die Ergebniswerte aller Knoten einer Schicht schrittweise partiell Berechnet. Für alle bis zu 450 Knoten einer Schicht werden die Ausgabewerte des ersten Knotens der vorhergehenden Schicht Gewichtet und im entsprechenden Register aufsummiert. Anschließend geschieht das gleiche für den zweiten Knoten der vorhergehenden Schicht, bis alle Ausgabewerte der Knoten der vorhergehenden Schicht gewichtet und zur aktuellen Netzeingabe aufsummiert wurden. Das bedeutet, das 450 Register in jeder Iteration geladen, für eine Addition verwendet, und wieder beschrieben werden. Da die Netzeingabe eines jeden Knotens einer Schicht somit erst berechnet wurde, sobald die Berechnung der Netzeingabe einer Schicht komplett abgeschlossen ist, können Ergebniswerte nicht frühzeitig an die nächste Berechnung weitergegeben werden.

7.3 Bewertung von SDSoC

Die Entwicklung von Hardware/Software Co-Designs in SDSoC ist Anfangs sehr schwierig. Es ist zunächst unklar, wie genau die Hardwarestrukturen aus dem Design in SDSoC realisiert werden können. Die Dokumentation von SDSoC ist ausreichend, jedoch muss oftmals ergänzend die Dokumentation von HLS gelesen werden. Diese Dokumentation ist sehr Umfangreich und die Durcharbeitung und Verinnerlichung der darin beschriebenen Verfahren ist sehr zeitintensiv. Abgesehen davon ist der Entwickler unter Umständen dazu gezwungen, Komponenten des Design teilweise in einer Hardwarebeschreibungssprache zu beschreiben, und als vorgefertigte Plattform oder sogenannte C-Callable IP-Cores

in das SDSoC Projekt zu integrieren. Das ist z.B. der Fall, wenn Ein- und Ausgabeanschlüsse des FPGA verwendet werden. Die Programmierung von parallelen operierenden Hardwareschaltkreisen ist in einer Programmiersprache, die für die Ausführung auf sequentiellen Prozessoren entwickelt wurde, sehr irritierend und gewöhnungsbedürftig. Nach einer Eingewöhnungszeit von einigen Wochen sollte das Umdenken jedoch bereits stattgefunden haben. Sobald ein Grundverständnis über SDSoC und HLS aufgebaut wurde, ist die Entwicklung in SDSoC zeitlich gesehen sehr effizient. Die Entwicklungszeiten für Hardwarebeschleuniger sind im Vergleich zu der Entwicklung derer in Hardwarebeschreibungssprachen extrem kurz, der C-Quellcode ist überschaubar und schnell änderbar. Dafür verliert der Entwickler die Kontrolle darüber, wie das Hardwaredesign genau Funktioniert. Der Entwickler kann sich lediglich darauf verlassen, dass die gewünschte Funktionalität abgebildet wird, jedoch nicht wie genau diese abgebildet wird. Anschließende Feinarbeiten am generierten VHDL/Verilog Code sind annähernd Unmöglich, da der Hardwarebeschreibungscode für Menschen fast unlesbar ist. Das liegt daran, das Variablenbezeichnungen (Signal- und Registerbezeichnungen) nicht eindeutig die Funktion repräsentieren und das sehr viele Zustandsautomaten und Multiplexer generiert werden. Ein Entwickler, der sowohl in SDSoC als auch in Hardwarebeschreibungssprachen gut ist, wird mit Hardwarebeschreibungssprachen immer einen höheren Durchsatz und höhere maximale Taktraten erzielen. Der Ressourcenverbrauch eines von SDSoC generierten Hardwaredesigns ist wesentlicher höher als bei einer Implementierung in einer Hardwarebeschreibungssprache. Wie groß der Unterschied ist hängt von dem Design ab. Mit SDSoC wird auf Kosten von Qualität und Kontrolle eine höhere Quantität erzielt. Auf Grund der hohen Kosten der Entwicklungsumgebung wird SDSoC vermutlich überwiegend in der Industrie verwendet, was die Entwicklung einer offenen Moduldatenbank unwahrscheinlich erscheinen lässt. Zusammenfassend eignet sich SDSoC für die Beschleunigung von rechenintensiven Operationen am besten und ist eine gute Wahl, wenn die Implementierung nicht viel Zeit beanspruchen soll.

7.4 Fazit

SDSoC ist für die Entwicklung von Hardwarebeschleuniger geeignet und erreicht zufriedenstellende Ergebnisse. Der Geschwindigkeitsgewinn durch die Implementierung des KNN in SDSoC ist gut, diese kann jedoch weiter optimiert werden. Der Ressourcenverbrauch der Implementierung ist relativ hoch. SDSoC ist für die genannten Aufgaben geeignet.
Der niedrige Stromverbrauch in Kombination mit der schnellen Berechnung der Ergebnis-

se ist einer der Hauptgründe dafür, FPGA anstatt GPU für das Testen von KNN zu verwenden. Die Auslagerung von gewöhnlichen C Code auf die Hardware ist möglich, verspricht jedoch keinen Leistungsgewinn. Für die Maximierung des Durchsatzes ist tieferes Wissen in SDSoC / HLS und unter Umständen einer Hardwarebeschreibungsprache notwendig. SDSoC ist seit 2015 auf dem Markt und funktioniert dafür bereits ziemlich zuverlässig. Die Komplexität der gesamten Produktkette die darin und damit verwendet wird ist extrem hoch. Die Kosten belaufen sich für eine „Node-Locked License" (an einen Rechner gebundene Lizenz) für SDSoC alleine auf $995. Um SDSoC vollständig nutzen zu können und Hardwarebeschreibungscode zu generieren, wird zusätzlich Vivado HL System Edition benötigt, was weiterhin $3495 kostet. Vermutlich wird auf Grund der Kosten der Großteil der in SDSoC entwickelten Hardware/Software Co-Designs nur intern in Unternehmen und Hochschulen vorhanden und geteilt werden und sich keine oder nur kleine offene Gemeinschaft bilden. Eine freie Alternative zur Beschreibung von Hardwaredesigns und Hardware/Software Co-Design in einer höheren Programmiersprache bieten z.B. die Sprachen SpinalHDL und Chisel. Wenn genügend Entwickler ein Interesse für diese Sprachen entwickeln, könnte es dazu kommen, dass eine Sammlung von verifizierten Designvorlagen entsteht. Darunter könnten auch unterschiedliche Varianten von neuronalen Netzen sein. Um die Qualität dieser Designvorlagen zu sichern, wird zusätzlich eine Prüfinstanz mit sehr guten Entwicklern benötigt, so wie es im Open-Source Projekt Linux der Fall ist.

Kapitel 8

Literaturverzeichnis

[Arm11] ARM: AMBA AXI and ACE Protocol Specification / Arm. 2011. – Forschungsbericht

[BFW95] BRASPENNING, P. J. Petrus J. (Hrsg.) ; F., Thuijsman (Hrsg.) ; WEIJTERS, A. J. M. M. (Hrsg.): *Artificial neural networks : an introduction to ANN theory and practice*. Berlin, New York : Springer, 1995 (Lecture notes in computer science). http://opac.inria.fr/record=b1090507. – ISBN 3–540–59488–4

[Ert08] ERTEL, Wolfgang: *Grundkurs Künstliche Intelligenz Eine praxisorientierte Einführung*. 1. Friedr. Vieweg und Sohn Verlag | GWV Fachverlage GmbH, 2008. – ISBN 978–3–528–05924–8

[GAGN15] GUPTA, Suyog ; AGRAWAL, Ankur ; GOPALAKRISHNAN, Kailash ; NARAYANAN, Pritish: Deep Learning with Limited Numerical Precision. In: *CoRR* abs/1502.02551 (2015). http://arxiv.org/abs/1502.02551

[GBB11] GLOROT, Xavier ; BORDES, Antoine ; BENGIO, Yoshua: Deep Sparse Rectifier Neural Networks. In: GORDON, Geoffrey J. (Hrsg.) ; DUNSON, David B. (Hrsg.): *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)* Bd. 15, Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011, 315-323

[GS98] GOKHALE, M. B. ; STONE, J. M.: NAPA C: compiling for a hybrid RISC/FPGA architecture. In: *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, 1998, S. 126–135

[JPJ08] JANG, H. ; PARK, A. ; JUNG, K.: Neural Network Implementation Using CUDA and OpenMP. In: *Digital Image Computing: Techniques and Applications (DICTA), 2008*, 2008, S. 155–161

[KBB+15] KRUSE, Rudolf ; BORGELT, Christian ; BRAUNE, Christian ; KLAWONN, Frank ; MOEWES, Christian ; STEINBRECHER, Matthias: *Computational Intelligence Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. 2. Wiesbaden : Springer Vieweg, 2015. http://dx.doi.org/10.1007/978-3-658-10904-2. http://dx.doi.org/10.1007/978-3-658-10904-2. – ISBN 978–3–658–10903–5

[Kra09] KRAMER, Oliver: *Computational Intelligence: Eine Einführung*. Dordrecht : Springer, 2009 (Informatik im Fokus). http://dx.doi.org/10.1007/978-3-540-79739-5. http://dx.doi.org/10.1007/978-3-540-79739-5. – ISBN 978–3–540–79738–8

[KSH12] KRIZHEVSKY, Alex ; SUTSKEVER, Ilya ; HINTON, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: PEREIRA, F. (Hrsg.) ; BURGES, C. J. C. (Hrsg.) ; BOTTOU, L. (Hrsg.) ; WEINBERGER, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, S. 1097–1105

[LHCS14] LOUSE H. CROCKETT, Martin A. E. Ross A. Elliot E. Ross A. Elliot ; STEWART, Robert W.: *The Zynq Book. Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, 2014

[MH95] MIRA, José (Hrsg.) ; HERNÁNDEZ, Francisco S. (Hrsg.): *From Natural to Artificial Neural Computation, International Workshop on Artificial Neural Networks, IWANN '95, Malaga-Torremolinos, Spain, June 7-9, 1995, Proceedings*. Bd. *930*. Springer, 1995 (Lecture Notes in Computer Science). – ISBN 3–540–59497–3

[MP43] MCCULLOCH, Warren ; PITTS, Walter: A Logical Calculus of Ideas Immanent in Nervous Activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 127–147

[NSP+16] NANE, R. ; SIMA, V. M. ; PILATO, C. ; CHOI, J. ; FORT, B. ; CANIS, A. ; CHEN, Y. T. ; HSIAO, H. ; BROWN, S. ; FERRANDI, F. ; ANDERSON, J.

; BERTELS, K.: A Survey and Evaluation of FPGA High-Level Synthesis Tools. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP (2016), Nr. 99, S. 1–1. http://dx.doi.org/10.1109/TCAD.2015.2513673. – DOI 10.1109/TCAD.2015.2513673. – ISSN 0278–0070

[Nvi16] NVIDIA: Datasheet: Nvidia® Tesla® P40 - Inferencing Accelerator / Nvidia. 2016. – Forschungsbericht

[OR06] OMONDI, Amos R. ; RAJAPAKSE, Jagath C.: *FPGA Implementations of Neural Networks*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2006. – ISBN 0387284850

[RW08] REY, Günter D. ; WENDER, Karl F.: *Neuronale Netze: Eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. Bern : Huber, 2008. – 207 S.. – ISBN 978–3–456–84513–5, 3–456–84513–8

[Rü96] RÜCKERT, Ulrich: Hardwareimplementierung Neuronaler Netze. In: *Konnektionismus und Neuronale Netze Beiträge zur Herbstschule (HeKoNN96)*, 1996, S. 53–64

[Sau10] SAUER, Peter: *Hardware-Design mit FPGA*. Elektor Verlag, 2010. – ISBN 9783895762093

[TG01] TANENBAUM, Andrew S. ; GOODMAN, James R.: *Computerarchitektur - Strukturen, Konzepte, Grundlagen (4. Aufl.)*. Pearson Studium, 2001. – ISBN 978–3–8273–7016–7

[Wan98] WANNEMACHER, Markus: *Das FPGA-Kochbuch*. International Thomson Publishing, 1998. – ISBN 3826627121

[Xil14a] XILINX: 7 Series DSP48E1 Slice (UG479) / Xilinx. 2014. – Forschungsbericht

[Xil14b] XILINX: 7 Series FPGAs Configurable Logic Block (ug474) / Xilinx. 2014. – Forschungsbericht

[Xil14c] XILINX: 7 Series FPGAs Memory Resources (ug473) / Xilinx. 2014. – Forschungsbericht

[Xil15] XILINX: Zynq-7000 All Programmable SoC Technical Reference Manual (UG585) / Xilinx. 2015. – Forschungsbericht

[Xil16a] XILINX: SDSoC Environment: User Guide (UG1027) / Xilinx. 2016. – Forschungsbericht

[Xil16b] XILINX: Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide (UG953) / Xilinx. 2016. – Forschungsbericht

[Xil16c] XILINX: Zynq-7000 All Programmable SoC Overview (DS190) / Xilinx. 2016. – Forschungsbericht